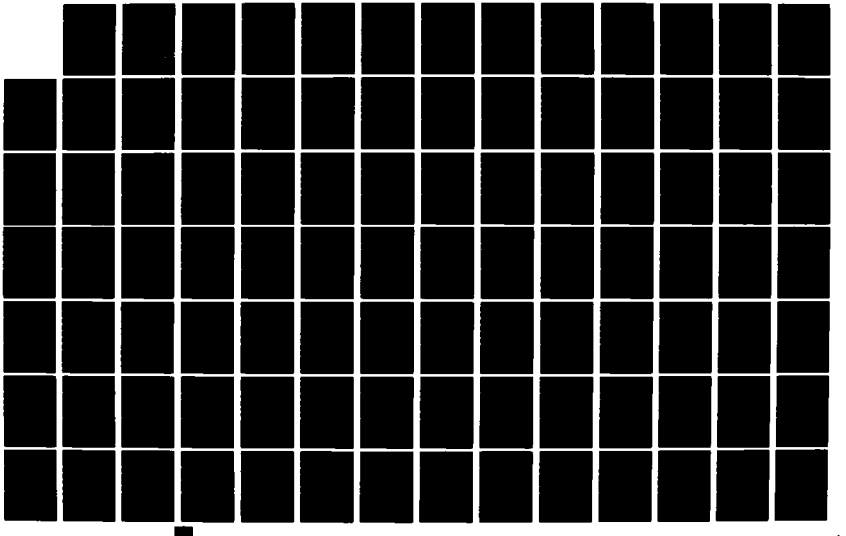
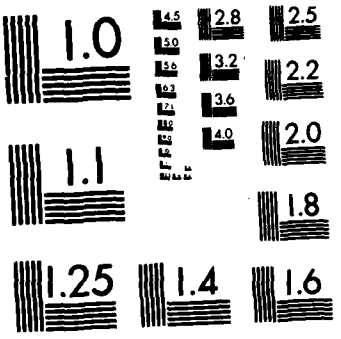
AD-A124 853 APPLICATIONS DIRECTED MICROPROGRAMMING ON A MINICOMPUTER SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 1/2

UNCLASSIFIED G A SCHOON DEC 82 AFIT/GCS/EE/82D-31 F/G 9/2 NL





DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/EE/82D-31

APPLICATIONS DIRECTED MICROPROGRAMMING ON A MINICOMPUTER SYSTEM

THESIS

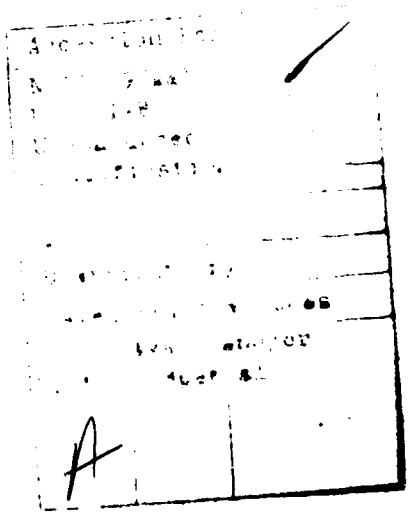
AFIT/GCS/EE/82D-31 GARY A. SCHOON
Capt USAF

AFIT/GCS/EE/82D-31

APPLICATIONS DIRECTED MICROPROGRAMMING ON A MINICOMPUTER SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science



by
GARY A. SCHOON, B.S.
Capt USAF
Graduate Computer Systems
December 1982

Acknowledgements

I would like to thank my thesis advisor, Dr. Gary Lamont for his guidance throughout this project. I would also like to thank Dr. Thomas Hartrum and Major Charles Lillie for serving on my thesis committee.

I would like to thank Mr. John Steidle for getting me started on the project and acting as my sponsor.

Mr. Glenn Williams and Mr. Conrad Phillippi deserve special thanks for volunteering the use of their computer programs and for assisting me in understanding and using the programs.

Finally, I wish to thank my wife Malee and daughter Michelle for their patience during this difficult period.

Contents

Page

List of Figures

List of Tables

Abstract

The use of microprogramming to improve the performance of application programs was investigated. The application programs used in the study were from various research laboratories at Wright-Patterson Air Force Base, Ohio. The user-microprogrammable Hewlett-Packard (HP) 21MX minicomputer was used for the investigation.

Two application programs were chosen as candidates for microprogramming, a wind tunnel stress analysis program and a laser materials modeling program. The programs were analyzed to determine where microprogramming should be applied using an activity profile generator program. The microcode for the programs was implemented, and the speed improvement measurements of the resultant programs were made.

The study further looked at the feasibility of automating the microprogramming tuning process on the HP 21MX computer. Approaches to automatically selecting program segments for microprogramming and automatically synthesizing the microcode were discussed.

Applications Directed Microprogramming on a Minicomputer System

I. Introduction

Introduction

General purpose computers are by definition designed to be used for a wide variety of applications, and thus are very versatile. It is because of this versatility, however, that these computers are inherently ineffecient for many applications. The ideal situation, from a performance point of view, would be to have a computer which was designed specifically for each application. Since this is not realistic, the user must usually accept the performance of the general purpose computer. For most applications, this is quite acceptable.

Some applications, however, may have requirements which exceed the capability of the general purpose computer. The user may then be forced to buy a special purpose machine -- a very expensive solution to the performance problem. If, however, the user's general purpose machine is user-microprogrammable, another possible solution exists. The user-microprogrammable computer can often be "tuned" using microprogramming to meet the specifications of special application programs. That is, special instructions can be added to the computer's instruction set which will more efficiently perform the basic operations or

"primitives" of the application program. T.G. Rauscher notes: "The efficiency of solving a particular problem depends primarily on the degree to which the architecture supports the problem primitives" (Ref. 1:1006).

It is this use of microprogramming to improve the performance of application programs which is the subject of this thesis investigation. This introductory chapter covers background information, the specific problem investigated, and the approach taken to solve the problem.

Background

The origin of microprogramming can be traced back to 1951 when M.V. Wilkes (Ref. 2) proposed using "microprogrammes" as an alternative to the "ad hoc manner" in which computer control units were being designed. The technique was not widely used commercially until the mid 1960s when IBM introduced the microprogrammed version of the System/360 (Ref. 3). Since then microprogramming has been widely used in the design of computer control units.

The introduction of a writable control store (WCS), that is read/write memory used to store microprograms, made it practical to use microprogramming to improve the performance of application programs. Depending on the application, performance can mean such things as speed, accuracy, or special data formats (Ref. 4:25). Speed is the primary performance measure considered here. Properly applied, microprogramming can increase the speed of a program

considerably. Gains of six to ten times or more are possible (Ref. 5:98). Because of limited memory available for microprograms and because of the complexity of the microprogramming task, it is not possible to completely microprogram most application programs. Microprogramming is therefore applied at points in the application program where most of the execution time is spent. For a more detailed discussion of microprogramming and how it provides improvement in program execution time, the reader should refer to Appendix A. Appendix B contains a glossary of terms used in this report.

Problem

The problem considered in this thesis investigation is the use of microprogramming on the user-microprogrammable Hewlett-Packard (HP) 21MX computer. This machine is used in several of the laboratories at Wright-Patterson Air Force Base (WPAFB) for a variety of specialized applications. Currently, little or no use is made of the microprogramming capabilty of the machine.

Previous work at the Air Force Institute of Technolgy (AFIT) on this problem was done by John J. Steidle (Ref. 6). Steidle implemented the user-microprogramming capability on the AFIT Digital Engineering Laboratory (DEL) HP 21MX and began the study of applying microprogramming to application programs. He was able to complete one microprogram -- a bit-reversal routine for a Fast Fourier

Transform (FFT) program. This thesis effort is essentially a continuation of his work.

Scope

The major objective of this thesis effort is to promote the use of user-microprogramming by:

1. Demonstrating its benefits in actual working application programs.
2. Investigating approaches which will aid other users in future microprogramming tuning efforts.

The result of this and future efforts will hopefully be improved program performance and extension of the useful life of the HP 21MX computer.

Approach

The approach taken in this thesis investigation is outlined in the following steps:

1. A literature search.
2. Identification of existing application programs which could benefit from microprogramming.
3. Analysis of those programs to determine where microprogramming should be applied.
4. Design and implementation of the microprogrammed routines.
5. Analysis of the resulting programs.

6. Investigation of approaches which would simplify the tuning process.

A literature search was conducted to gain necessary background and to learn what related work had been done. The search revealed that research had been done in both manual (Refs. 1,4-12) and automatic (Refs. 13-19) techniques of architecture tuning.

Identification of candidate programs was accomplished by contacting HP users on base. An initial list of users was already available (Ref. 6:Appendix C). Users were interviewed to determine what application programs were available and which of these would make good candidates for microprogramming. Source code of selected programs was then obtained for further analysis.

Analysis of the selected programs was done using an activity profile generator program (Ref. 6:22). This program monitors the execution of an application program, and generates a table and a histogram showing the relative execution times of the various routines of that program.

The analysis of the programs identified potential routines for microprogramming. The microroutines were then designed, coded and substituted back into the original programs. The resultant programs were analyzed, and the execution times were compared with the execution times of the original programs.

Based on the experience gained through the above manual tuning process and work of others found in the litera-

ture search, the investigation of approaches to simplify the process was begun. The goal of this effort was to investigate the feasibility of completely automating the process and developing an automatic tuning system for the AFIT HP 21MX computer. Each step of the process was studied to determine if it could be automated, and the availability of software and algorithms to support the tuning step was examined.

Limitations

This thesis investigation was limited in several areas because of various factors. The study was confined to applications of microprogramming on the HP 21MX, although the concepts could be applied to any user-microprogrammable computer. The number of application programs tuned was limited by the number of potential programs identified by the base users and by the time frame of the thesis effort. The size of the microprograms was limited by the size of the WCS on the AFIT machine -- 256 words.

Order of Presentation

This report consists of seven chapters. Chapter I provides an intoduction and outlines the problem considered and the approach taken to solve that problem. Chapter II covers the survey of HP users and the two application programs found as candidates for microprogramming. The analysis of the two application programs is described in Chap-

ter III. Chapter IV describes the requirements, design, implementation, and test of the first microprogram -- a matrix multiplication routine used for stress calculations in a wind tunnel control program. The requirements, design, implementation and test of the microcode for the second candidate program -- a laser materials modeling program -- is covered in Chapter V. Chapter VI discusses the feasibility of designing an automated tuning system for the AFIT HP 21MX computer. Chapter VII presents the results, conclusions, and recommendations of the thesis investigation.

II. Survey of HP Users at Wright-Patterson Air Force Base

Introduction

In order to identify existing application programs which might benefit from microprogramming, a survey of HP users at Wright-Patterson Air Force Base was conducted. This chapter reports the details of this survey -- the organizations surveyed, the criteria for choosing candidate programs, and the candidate programs chosen for further analysis.

Organizations Surveyed

The survey was conducted through telephone contacts and personal interviews with HP users whose names appeared on an existing list (Ref. 6: Appendix C). An updated list is given in Appendix C. Each user contacted was given a brief explanation of the microprogramming tuning process and then asked if their organization had any programs which might benefit from this process.

Users from eight separate organizations at Wright-Patterson were surveyed. All of the organizations have at least one HP 21MX computer; one organization has four. The major uses of the HP 21MX differ widely among organizations, some of the uses being: sensor modeling, electronic warfare analysis and modeling, materials modeling, instru-

ment data acquisition and processing, on-line data acquisition of real-time telemetry data, wind tunnel control, and random vibration control.

Because of the diverse applications of the HP 21MX, the application programs of the various organizations have very little in common at the detailed level. At a more general level, however, the programs can be divided into three major categories -- modeling, data acquisition, and control.

Criteria for Candidate Programs

Because of the large number of application programs being run on the HP 21MX, some criteria had to be used in selecting programs for further analysis for microprogramming. Meyers (Ref. 20:29) suggests three criteria for determining whether a function should be implemented in microcode or software: (1) "the function should be small," (2) "the function should be unlikely to change, and" (3) "system performance would suffer from a slower software implementation of the function." Although Meyers is applying these criteria to the design of computer architectures, they are also very applicable to the "tuning" process considered here, and thus were used in the program selection process.

The requirement that the function be small is necessary for two reasons. Writable control store is very limited on most user-microprogrammable computers. The AFIT HP

21MX, one of the HP 1000 Series computers, for example, has only 256 words. Table I (Ref. 7:15) shows the control store options available for the HP 1000 Computer Series. Also, microprogramming is inherently more difficult than programming in a higher level language, because of the lower-level details the programmer must be concerned with, such as register and bus transfers, arithmetic logic unit (ALU) operations, and microinstruction timing. For this reason microprogramming should be held to a minimum.

The two reasons for the first criterion also apply to the second, that "the function should be unlikely to change." Limited writable control store makes program growth difficult, if not impossible in many cases. The complexity of microprogramming makes the modifications much more expensive.

The third criterion points to the program's need for performance improvement. This may be the most important of the three criteria. If a user feels the performance of a program is already adequate, there is no need to add expense and complexity to it by adding microcode.

One additional criterion that should be considered is a program's potential for improvement using microprogramming. This potential is based on the nature of the program. A compute-bound program is more likely to be improved by microprogramming than an I/O-bound program. A plotting program that spends the majority of its execution time waiting for the mechanical plotter would gain nothing

TABLE I
User Microprogrammability Provisions of HP 1000 Computers

HP 1000 Computer Series		M	E	F
HP 1000 Computer Models	With 4 I/O channels	2105		
	With 9 I/O channels	2108	2109	2111
	With 14 I/O channels	2112	2113	2117
Control Store Space (micro-instructions)				
Total control store address space		4096	16384	16384
Space used by base instruction set		1024	1024	2816
Space reserved for HP enhancements		1536	3584	7936
Space reserved for user microprograms		1536	11776	5632
Control Store Hardware for the User				
12945A 256-instruction User Control Store board for user-installed ROMs		max. of 2	n/a	n/a
13407A 2048-instruction User Control Store board for user-installed ROMs		max. of 1	max. of 1	max. of 1
13197A 1024-instruction Writable Control Store board		max. of 2	max. of 3	max. of 3

in performance from microprogramming.

Candidate Programs

Applying these four criteria to programs examined in the survey, two application programs from two different organizations were chosen for further analysis -- a wind tunnel stress control program and a laser materials modeling program. The background and general requirements of these programs is dicussed here.

Wind Tunnel Stress Control Program. The wind tunnel stress control program, called STRES, is one of several subroutines used to control the overall operation of a 9-inch experimental wind tunnel used by one of the Air Force Weapons Laboratory (AFWAL) organizations -- AFWAL/FIMN (Ref. 21). STRES is used to calculate the stresses on flexible rods or elements in the wind tunnel.

A cross section of the tunnel with the rods is shown in Figure 1 (Ref. 22:Figure 3). A total of eighteen rods form the floor and ceiling of the tunnel, nine rods on each surface. Each of the rods is connected to ten electro-mechanical jacks, which are used to bend the flexible rods to a desired shape. Bending each of the rods provides the tunnel with variable geometry walls, which allows "testing larger models than previously possible in a comparable sized conventional tunnel" (Ref. 22:1). Figure 2 (Ref. 22:Figure 2) illustrates the effect that bending the rods has on the shape of the tunnel.

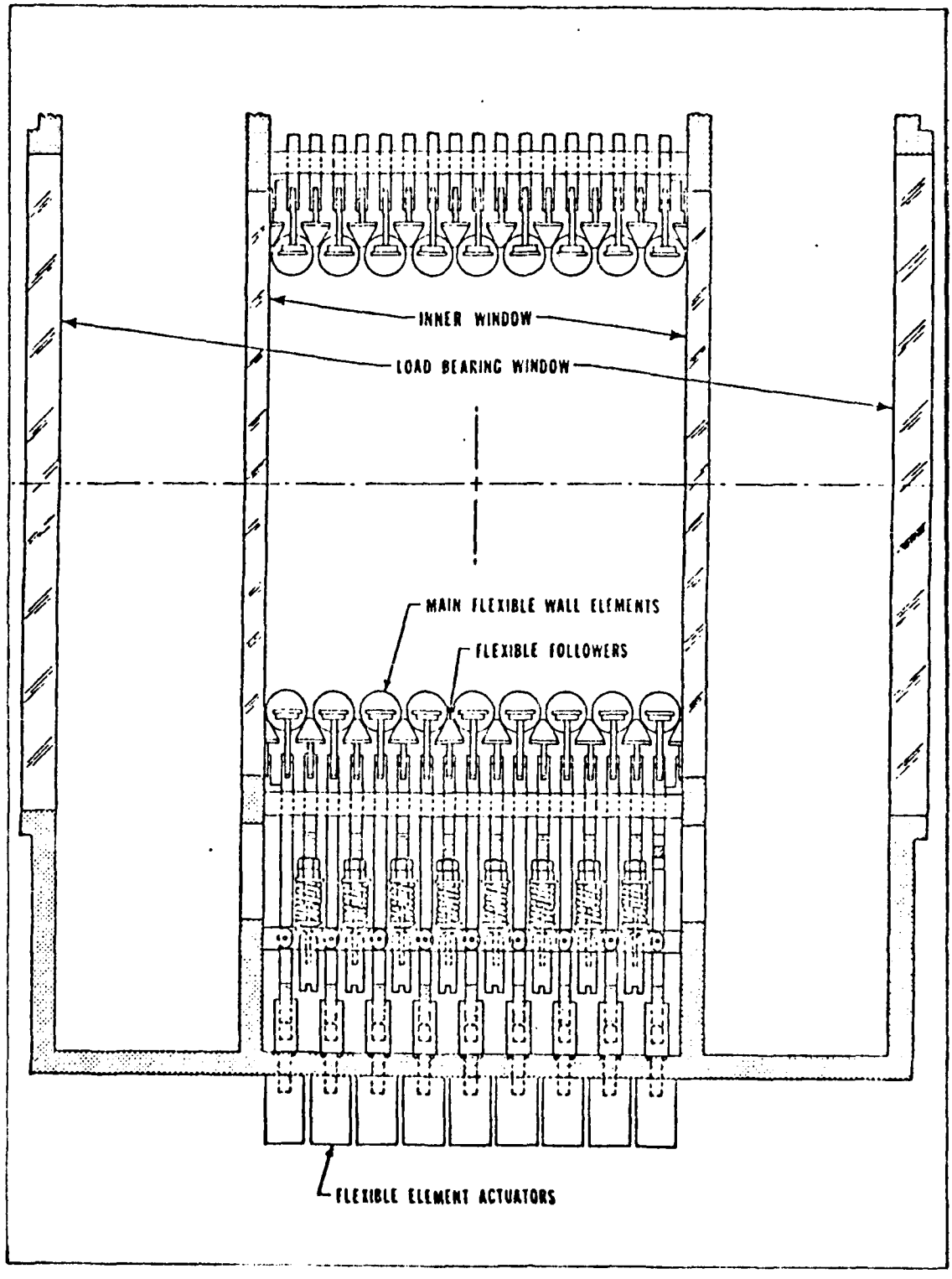


Figure 1. Wind Tunnel Cross Section

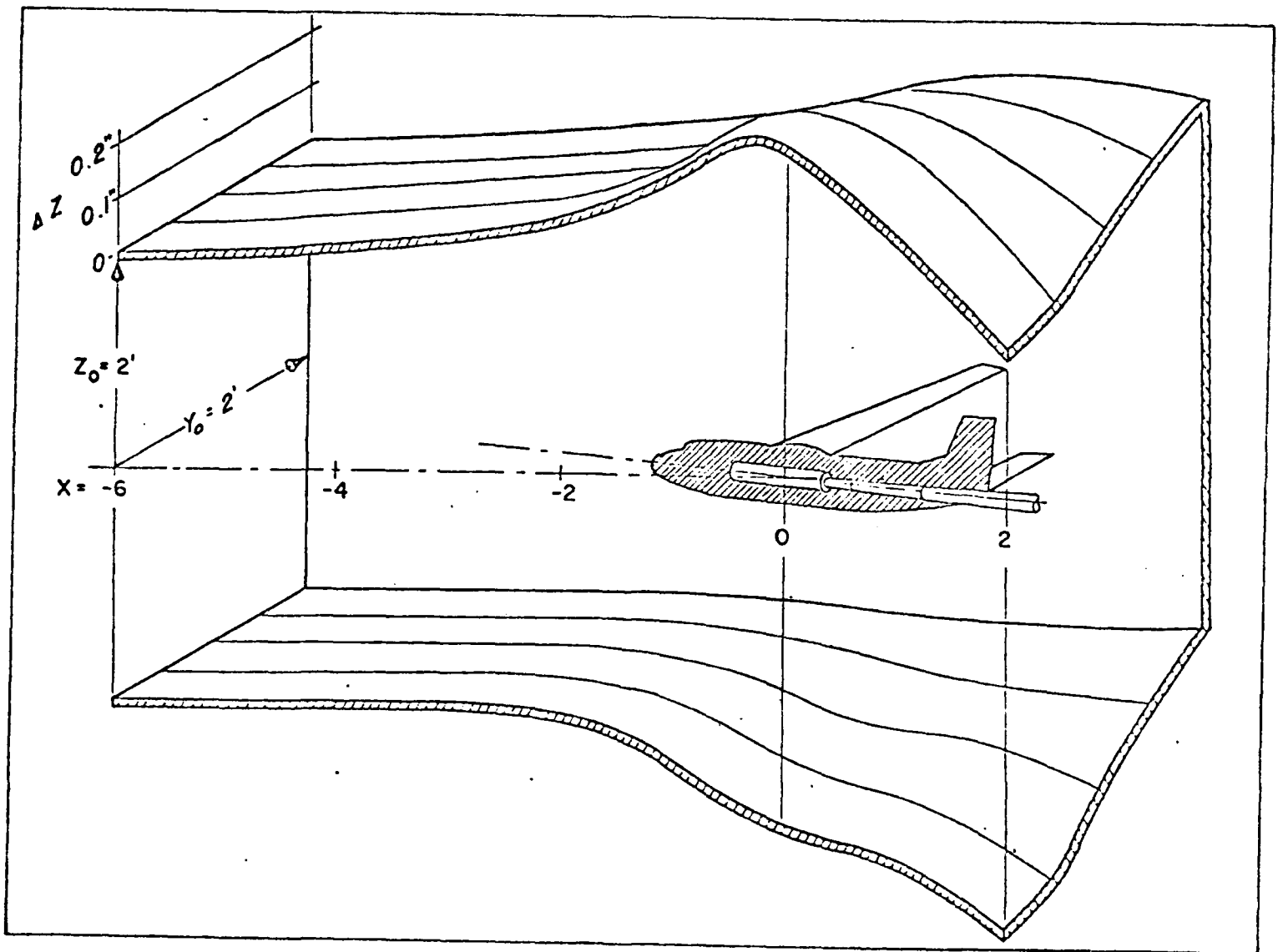


Figure 2. Wind Tunnel with Variable Geometry Walls

The rods are adjusted each time the tunnel is prepared for a model test. Although the rods are flexible, it is important that they are not over-stressed or they will be permanantly distorted. The function of STRES is to prevent this from happening. STRES receives as one of its input parameters an array containing the relative distance each of the ten adjusting jacks has been moved. This information is used to calculate the stresses and moments on each rod. These values are then passed to another routine which automatically shuts down one or more jacks if the maximum allowable values are exceeded.

Originally STRES and its associated subroutines were written entirely in FORTRAN, but the program was too slow to allow adjusting of more than one rod at a time. Part of one routine was rewritten in assembly language, and the gain in speed allowed the adjusting of three rods at a time. This new routine was called, quite appropriately, SPEED.

The rod adjustment process took about 5 minutes. It was hoped that by microprogramming parts of STRES and SPEED, the stress calculations could be made fast enough to allow the simultaneous adjustment of more than three rods -- possibly two or three times as many -- and thus reduce the total adjustment time. The ultimate goal was to be able to adjust all eighteen rods at once! This would allow real-time adjustments during a test.

The stress calculation function met all of the program

selection criteria, and was considered a prime candidate for microprogramming. More conventional speed-up techniques such as reverting to assembly language had been tried. Microprogramming was a logical next step.

Laser Materials Modeling Program. The laser materials modeling program is a program which was developed by personnel at AFWAL/MLPJ (Ref. 23) to model the optical characteristics of laser materials. This program is used to calculate the real and imaginary parts of refractive index, an important measure of laser materials. This measure is given by the following equations (Refs. 23, 24:1327):

$$n^2-k^2=\varepsilon_0+\sum_i 4\pi\sigma_i\nu_i^2\,\frac{\nu_i^2-\nu^2}{(\nu_i^2-\nu^2)^2+\gamma_i^2\nu_i^2\nu^2}-x\,\frac{\nu_p^2}{\nu^2+\nu_r^2}$$

$$nk=\sum_i 2\pi\sigma_i\nu_i\,\frac{\gamma_i\nu_i\nu}{(\nu_i^2-\nu^2)+\gamma_i^2\nu_i^2\nu^2}+x\,\frac{\nu_r\nu_p^2}{\nu(\nu^2+\nu_i^2)}$$

where:

n = real part of the refractive index (dimensionless)

k = imaginary part of the refractive index (dimensionless)

ε_0 = short-wavelength dielectric constant (dimensionless)

σ_i = strength of resonance (dimensionless)

ν_i = frequency of resonance (cm^{-1})

ν = radiation frequency (cm^{-1})

γ_i = damping factor (dimensionless)

x = a weighting factor (dimensionless)

ν_p = plasma frequency (cm^{-1})

ν_r = relaxation frequency (cm^{-1})

The HP 21MX at this laboratory is equipped with a potentiometer board consisting of 32 potentiometers. Each potentiometer of the board can be adjusted to supply different voltage levels to analog-to-digital (A/D) converters. The A/D converters are connected to the HP 21MX, providing digitized inputs of the potentiometer settings. In the modeling program the potentiometers are used to provide the trial input parameters for the above equations. Values for n and k can then be calculated.

Another important measure of laser materials is reflectivity. The reflectivity R at normal incidence is given by the following equation (Ref. 24:1327):

$$R = \frac{(n-1)^2+k^2}{(n+1)^2+k^2} \quad \text{(n,k, and R are dimensionless)}$$

Once n and k are known, R can be calculated and displayed on an oscilloscope as a function of wavelength.

An experimental method of obtaining a plot of reflectivity is by using a spectrometer to measure a laser material sample. The coordinate points for this measured plot can then be input to the modeling program and displayed along with the calculated waveform. An example of what a display might look like is shown in Figure 3. By adjusting the potentiometers, the calculated plot can be adjusted to

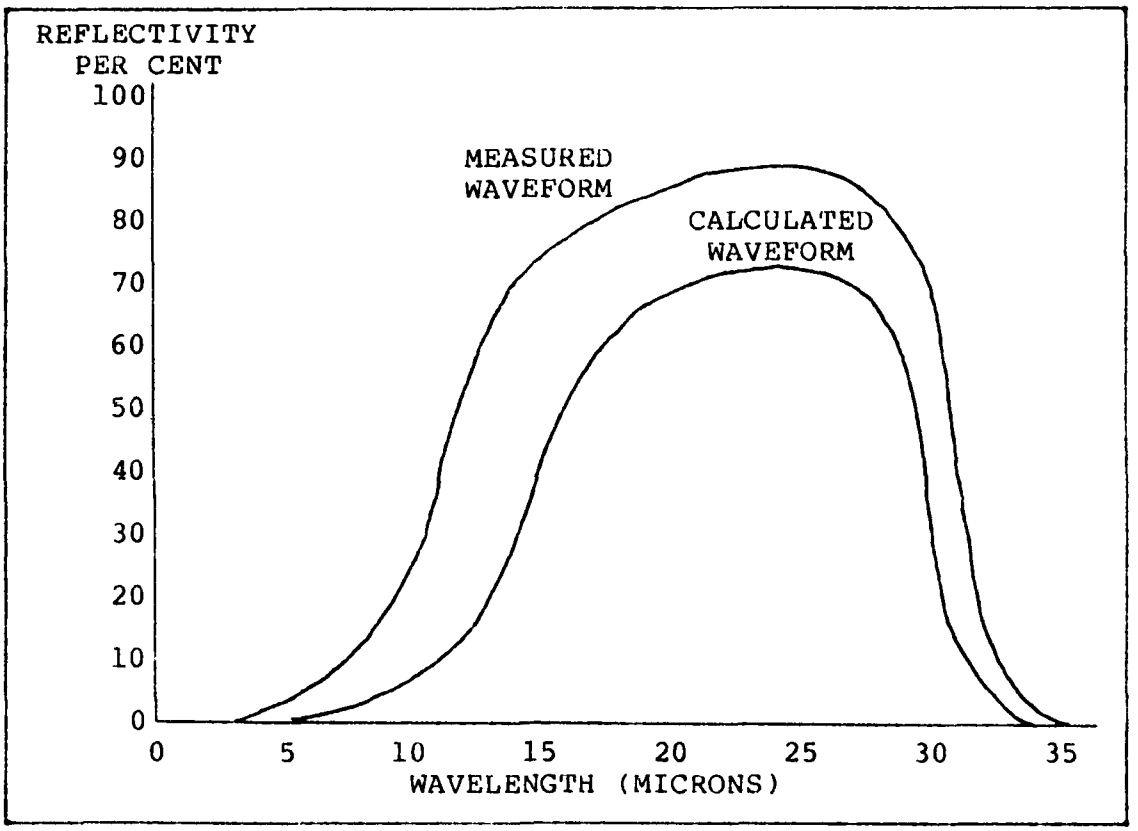


Figure 3. Reflectivity of a Laser Material Sample.

closely match the measured waveform and the corresponding refractive index spectra can be calculated. Thus, the program functions as a curve-fitting tool.

It should be made clear that there are other computerized techniques for calculating the refractive index, and that this technique is not meant to replace them. This experimental method has three objectives (Ref. 23): (1) to calculate the refractive index of a sample material, (2) to give a researcher a "feel" for the effects the different parameters have on the spectra, and (3) to possibly aid in the synthesis of new laser materials.

Although this modeling program had not yet been tested at the time of the survey, it was anticipated that the FORTRAN routine used to calculate the refractive index equations would not be fast enough to provide an acceptable oscilloscope display. This was the motivation behind the application of microprogramming to this routine.

The routine met all of the program selection criteria. The function was small. The refractive index equations were well established, so there was little possibility of modification. The function should execute faster in microcode. Since the program had not yet been tested, there was some question as to whether or not the function would execute fast enough in FORTRAN. The designer of the program had had experience with similar FORTRAN routines running on the HP 21MX, and felt confident that this routine would not be fast enough to provide an acceptable oscillo-

scope display if done in FORTRAN. Later tests confirmed that he was correct. Cosidering these factors, the program was a good candidate for microprogramming.

Conclusions

The results of this survey were somewhat unexpected, since only two application programs were selected as candidates for microprogramming. Several reasons for this can be given:

(1) Many programs did not meet the selection criteria.

(2) A Control Data Corporation (CDC) system is available to most organizations at Wright-Patterson Air Force Base and can be used for programs which exceed the capability of the HP computer.

(3) One of the organizations (Ref. 25) had potential programs, but administrative control of the computers was being transferred to another organization.

(4) One person interviewed (Ref. 26) had several ideas for microprogramming applications, but the programs had either not been written, or were running on the CDC system.

(5) Some persons may have been hesitant to involve themselves or their programs in this project.

Although only two application programs were identified, the survey was considered successful. The two programs were "real" operational programs and were good can-

didates for microprogramming. Also, the thinking of many of the HP users was hopefully stimulated toward the use of microprogramming in future applications.

III. Analysis of Candidate Programs

Introduction

As stated in the first chapter, it is not possible or even desirable to completely microprogram most application programs, because of limited writable control store and the complexity of the microprogramming task. Fortunately, microcoding an entire program is not necessary to increase its speed. One study of FORTRAN programs (Ref. 28) has shown that more than 80 per cent of the total execution time of a program is concentrated in at most four to five per cent of the instructions. Careful analysis of a program can reveal these areas of high concentration.

This chapter covers some of the analysis techniques and the application of one these techniques to the two microprogramming candidates -- the wind tunnel stress program and the laser materials modeling program.

Analysis Techniques

Several techniques are available for determining where most of the execution time is spent in a program (Ref. 4:146):

1) Static instruction analysis

2) Timing calls

3) Logic analyzer

4) Activity profile generator

Static instruction analysis involves the addition of the execution times of individual instructions in a program to determine the total execution times of the various segments of the program. This method can provide good results if the execution is not data-dependent. Detailed knowledge of the individual instruction execution times is required, however, and the process is very tedious if done manually.

Timing calls to the system clock can be used to determine the elapsed time between the beginning and end of a program segment. This technique requires a high-resolution system clock. Erroneous results may be obtained in a multitasking system. Also, there is much guesswork involved in the placement of the timing calls within the program.

A logic analyzer can be used to monitor memory accesses. If the absolute addresses of the program segments are known, the logic analyzer can be programmed to monitor those addresses, and the frequency of a segment's execution can be determined. This is an good technique, because it gives very precise measurements without any interference with the operation of the computer. It does require the added hardware of the logic analyzer, and like the timing call technique, involves much guesswork in determining the program addresses to monitor.

The activity profile generator is a program which runs in a multitasking system along with the program under test. The profile generator uses an external interrupt, such as

the system clock, to interrupt the program under test, and the point of interrruption is recorded. These recorded points of interruption can then be used to generate a table or histogram of the program's activity, showing immediately the most active segments of the program. This technique has the advantage of easy implementation on most systems. It does introduce some overhead, however, because of the frequent interruption of the program under test, but the results of the profile generation are not affected. Also, because the profile generator runs in a multitasking system, erroneous results may be obtained as in the timing call method. This problem can be solved by the correct setting of program execution priorities.

One additional technique that has been used (Refs. 17, 29) is a microprogrammed version of the activity profile generator. This technique requires modifications to the microprogrammed instruction fetch routine to gather the statistics on instructions as they are fetched from memory for execution. Since the fetch routine "sees" each instruction, a detailed execution profile can be made. The detail can be down to the number of times each program instruction is executed, if desired. Some overhead is introduced, since the instruction fetch time is increased. This technique is difficult to implement on most commercial machines, since it requires modification of the instruction fetch routine in control store ROM. Also, some provision is needed for turning the profile off under normal opera-

tion of the machine.

Activity Profile Generator Program

Of the analysis techniques discussed, the activity profile generator program was chosen for this study. Static instruction analysis was ruled out because it would have had to be done manually. The timing call technique would have involved too much guesswork, especially in the unfamiliar application programs being analyzed. A logic analyzer was not available, so was not seriously considered. Modification of the instruction fetch routine on a commercial machine is something which should be done by the manufacturer rather than the user. Because of these reasons, the activity profile generator program was considered the best choice. Also, one such program called ACTV was available, and had successfully been used before (Ref. 6:24).

A listing of the program ACTV and its two subroutines is given in Appendix D. The subroutine called IDGET was added to the original program to allow ACTV to run on AFIT's RTE-III (Real-Time Executive-III) operating system. This routine is available as a system library routine on RTE-IV systems. Instructions for running ACTV under RTE-III are given in Appendix E.

ACTV monitors a program's activity by periodically interrupting the program, using the system clock as the source of the interrupt. The interrupt rate in increments

of ten milliseconds is interactively input by the user. The user also provides an upper and lower bound on the memory addresses of interest based on the load address of the program. This area of interest is divided into 50 intervals, and a counter is provided for each interval in the form of an array. Two additional counters are used -- one for addresses below the area of interest and one for addresses above. The address at which the program is suspended at the time of interruption determines which counter is incremented. The number of counts or "hits" in each address interval can then be printed in the form of a table and a histogram providing the activity profile for the program.

Wind Tunnel Stress Program Analysis

The analysis of the wind tunnel stress calculation program STRES and its subroutine SPEED was performed using the profile data from several computer runs of ACTV. In order to run STRES on the AFIT HP system, a special test driver program called SDRVR was obtained from AFWAL/FIMN. This program provided the needed input parameters to STRES which normally originate from special hardware of the wind tunnel control system. For the analysis, SDRVR was used as the calling program for STRES, which in turn called SPEED. Neither STRES nor SPEED were modified for the analysis. Listings for the three programs are provided in Appendix F.

Tables II, III, and IV show the resulting activity

profile tables for three of the ACTV runs. ACTV also outputs histograms corresponding to the tables, but these were found to be of little value for the analysis, and are not shown. Table V is a load map showing the absolute memory addresses of SDRVR, STRES, and SPEED. The addresses from the load map are used to correlate the address intervals on the ACTV tables to the programs.

The first ACTV run looked at the memory addresses from 40531 (all addresses in octal) to 43126, which included SDRVR, STRES, and SPEED. Table II shows that 10 of the 14 total "hits" or 71 per cent of the activity occurred in SPEED.

The next run looked at addresses 42604 to 43126, the range of SPEED. Table III shows that 8 of the 10 "hits" in SPEED occurred in the address range of 42666 to 42705.

To further "home" in on the "hot" spot, a third run was made looking at the interval from 42661 to 43020. Table IV shows the 8 "hits" occurred in the range of 42671 to 42703 or locations 65 to 77 of SPEED.

These locations correspond very closely to the range of LOOP1 in SPEED. With 80 per cent of the activity of SPEED occurring in LOOP1, the obvious conclusion to be drawn is that any microprogramming applied to SPEED should include the LOOP1 program segment.

Laser Materials Modeling Program Analysis

The analysis of the laser materials modeling program

TABLE II

Program Activity Profile for SDRVR, STRES, and SPEED
From 040531 to 043126 in Increments of 26

INTERVAL NO.	FROM	TO	NO. OF HITS	NORMALIZED HITS	NORMAL ACCUM
1	000000	040531	.	.00000000	.00000
2	040531	040563	.	.00000000	.00000
3	040563	040615	.	.00000000	.00000
4	040615	040647	.	.00000000	.00000
5	040647	040701	.	.00000000	.00000
6	040701	040733	.	.00000000	.00000
7	040733	040765	.	.00000000	.00000
8	040765	041017	.	.00000000	.00000
9	041017	041051	.	.00000000	.00000
10	041051	041103	.	.00000000	.00000
11	041103	041135	.	.00000000	.00000
12	041135	041167	.	.00000000	.00000
13	041167	041221	.	.00000000	.00000
14	041221	041253	.	.00000000	.00000
15	041253	041305	.	.00000000	.00000
16	041305	041337	.	.00000000	.00000
17	041337	041371	.	.00000000	.00000
18	041371	041423	.	.00000000	.00000
19	041423	041455	.	.00000000	.00000
20	041455	041507	.	.00000000	.00000
21	041507	041541	.	.00000000	.00000
22	041541	041573	.	.00000000	.00000
23	041573	041625	.	.00000000	.00000
24	041625	041657	.	.00000000	.00000
25	041657	041711	.	.00000000	.00000

TABLE II (cont.)

26	041711 041743	.	.00000000	.00000
27	041743 041775	2.	.28571427	.14286
28	041775 042027	.	.00000000	.14286
29	042027 042061	.	.00000000	.14286
30	042061 042113	.	.00000000	.14286
31	042113 042145	.	.00000000	.14286
32	042145 042177	.	.00000000	.14286
33	042177 042231	.	.00000000	.14286
34	042231 042263	.	.00000000	.14286
35	042263 042315	.	.00000000	.14286
36	042315 042347	.	.00000000	.14286
37	042347 042401	.	.00000000	.14286
38	042401 042433	.	.00000000	.14286
39	042433 042465	2.	.28571427	.28571
40	042465 042517	.	.00000000	.28571
41	042517 042551	.	.00000000	.28571
42	042551 042603	.	.00000000	.28571
43	042603 042635	.	.00000000	.28571
44	042635 042667	1.	.14285713	.35714
45	042667 042721	7.	1.00000000	.85714
46	042721 042753	.	.00000000	.85714
47	042753 043005	1.	.14285713	.92857
48	043005 043037	1.	.14285713	1.00000
49	043037 043071	.	.00000000	1.00000
50	043071 043123	.	.00000000	1.00000
51	043123 043155	.	.00000000	1.00000
52	043155 077777	.	.00000000	1.00000

TABLE III

Program Activity Profile for SPEED
From 042604 to 043126 in Increments of 5

INTERVAL NO.	FROM	TO	NO. OF HITS	NORMALIZED HITS	NORMAL ACCUM
1	000000	042604	4.	1.00000000	.28571
2	042604	042611	.	.00000000	.28571
3	042611	042616	.	.00000000	.28571
4	042616	042623	.	.00000000	.28571
5	042623	042630	.	.00000000	.28571
6	042630	042635	.	.00000000	.28571
7	042635	042642	.	.00000000	.28571
8	042642	042657	.	.00000000	.28571
9	042657	042654	.	.00000000	.28571
10	042654	042661	.	.00000000	.28571
11	042661	042666	.	.00000000	.28571
12	042666	042673	4.	1.00000000	.57143
13	042673	042700	3.	.75000000	.78571
14	042700	042705	1.	.25000000	.85714
15	042705	042712	.	.00000000	.85714
16	042712	042717	.	.00000000	.85714
17	042717	042724	.	.00000000	.85714
18	042724	042731	.	.00000000	.85714
19	042731	042736	.	.00000000	.85714
20	042736	042743	.	.00000000	.85714
21	042743	042750	.	.00000000	.85714
22	042750	042755	.	.00000000	.85714
23	042755	042762	.	.00000000	.85714
24	042762	042767	.	.00000000	.85714
25	042767	042774	1.	.25000000	.92857

TABLE III (cont.)

26	042774	043001	.	.00000000	.92857
27	043001	043006	.	.00000000	.92857
28	043006	043013	1.	.25000000	1.00000
29	043013	043020	.	.00000000	1.00000
30	043020	043025	.	.00000000	1.00000
31	043025	043032	.	.00000000	1.00000
32	043032	043037	.	.00000000	1.00000
33	043037	043044	.	.00000000	1.00000
34	043044	043051	.	.00000000	1.00000
35	043051	043056	.	.00000000	1.00000
36	043056	043063	.	.00000000	1.00000
37	043063	043070	.	.00000000	1.00000
38	043070	043075	.	.00000000	1.00000
39	043075	043102	.	.00000000	1.00000
40	043102	043107	.	.00000000	1.00000
41	043107	043114	.	.00000000	1.00000
42	043114	043121	.	.00000000	1.00000
43	043121	043126	.	.00000000	1.00000
44	043126	043133	.	.00000000	1.00000
45	043133	043140	.	.00000000	1.00000
46	043140	043145	.	.00000000	1.00000
47	043145	043152	.	.00000000	1.00000
48	043152	043157	.	.00000000	1.00000
49	043157	043164	.	.00000000	1.00000
50	043164	043171	.	.00000000	1.00000
51	043171	043176	.	.00000000	1.00000
52	043176	077777	.	.00000000	1.00000

TABLE IV

Program Activity Profile for LOOP1 of SPEED
From 042661 to 043020 in Increments of 2

INTERVAL NO.	FROM	TO	NO. OF HITS	NORMALIZED HITS	NORMAL ACCUM
1	000000	042661	4.	1.00000000	.28571
2	042661	042663	.	.00000000	.28571
3	042663	042665	.	.00000000	.28571
4	042665	042667	.	.00000000	.28571
5	042667	042671	.	.00000000	.28571
6	042671	042673	2.	.50000000	.42857
7	042673	042675	4.	1.00000000	.71429
8	042675	042677	1.	.25000000	.78571
9	042677	042701	.	.00000000	.78571
10	042701	042703	1.	.25000000	.85714
11	042703	042705	.	.00000000	.85714
12	042705	042707	.	.00000000	.85714
13	042707	042711	.	.00000000	.85714
14	042711	042713	.	.00000000	.85714
15	042713	042715	.	.00000000	.85714
16	042715	042717	.	.00000000	.85714
17	042717	042721	.	.00000000	.85714
18	042721	042723	.	.00000000	.85714
19	042723	042725	.	.00000000	.85714
20	042725	042727	.	.00000000	.85714
21	042727	042731	.	.00000000	.85714
22	042731	042733	.	.00000000	.85714
23	042733	042735	.	.00000000	.85714
24	042735	042737	.	.00000000	.85714
25	042737	042741	.	.00000000	.85714

TABLE IV (cont.)

26	042741	042743	.	.00000000	.85714
27	042743	042745	.	.00000000	.85714
28	042745	042747	.	.00000000	.85714
29	042747	042751	.	.00000000	.85714
30	042751	042753	.	.00000000	.85714
31	042753	042755	.	.00000000	.85714
32	042755	042757	.	.00000000	.85714
33	042757	042761	.	.00000000	.85714
34	042761	042763	.	.00000000	.85714
35	042763	042765	.	.00000000	.85714
36	042765	042767	1.	.25000000	.92857
37	043767	042771	.	.00000000	.92857
38	042771	042773	.	.00000000	.92857
39	042773	042775	.	.00000000	.92857
40	042775	042777	.	.00000000	.92857
41	042777	043001	.	.00000000	.92857
42	043001	043003	.	.00000000	.92857
43	043003	043005	.	.00000000	.92857
44	043005	043007	.	.00000000	.92857
45	043007	043011	1.	.25000000	1.00000
46	043011	043013	.	.00000000	1.00000
47	043013	043015	.	.00000000	1.00000
48	043015	043017	.	.00000000	1.00000
49	043017	043021	.	.00000000	1.00000
50	043021	043023	.	.00000000	1.00000
51	043023	043025	.	.00000000	1.00000
52	043025	077777	.	.00000000	1.00000

TABLE V

Load Map for SDRVR, STRES, and SPEED

```
COM     40002 40530
SDRVR   40531 42110
STRES   42111 42603
SPEED   42604 43126

..MAP   43127 43222  751101  24998-16001
CLRIO   43223 43231  750701  24998-16001

ENTRY POINTS

 *SDRVR  41673
 *.DLD  104200
 *.DST  104400
 *..MAP  43127
 *EXEC   12446
 *CLRIO  43223
 *STRES  42244
 *.FMP  105040
 *.ENTR  37201
 *FLOAT 105120
 *SPEED  42611
```

subroutine called CALC followed the scheme that was used in the analysis of the wind tunnel program. A special driver program called CDRVR was required to provide the needed input parameters for the routine CALC. Listings for the two FORTRAN programs, CDRVR and CALC, are in Appendix I.

As with the previous program, three activity profiles were run. Tables VI, VII, and VIII show the resulting activity profile tables, and Table IX shows the load map for CDRVR and CALC.

The first run included both CDRVR and CALC in the area of interest, from location 40002 to 40651. Table VI shows a large concentration of "hits" in the address range from 40233 to 40354, somewhere in the middle of CALC. The number of "hits" in this area is 33 of the total 48, or 69 per cent.

The second run included CALC only from location 40142 to 40651. Table VII shows two areas of relatively high "hit" concentration. One large area from 40266 to 40347 contains 23 "hits" or 48 per cent of the total. The other smaller area from 40223 to 40250 has 8 "hits" or 17 per cent of the total.

The third run covered these two areas more closely from 40214 to 40365. Table VIII still shows the two areas of concentration, but within those areas, the "hits" are fairly evenly distributed. One exception is the interval from 40332 to 40335, which has 6 "hits", a relatively high concentration. A mixed FORTRAN/assembly listing shows that

TABLE VI

Program Activity Profile for CDRVR and CALC
From 040002 to 040651 in Increments of 9

INTERVAL NO.	FROM	TO	NO. OF HITS	NORMALIZED HITS	NORMAL ACCUM
1	000000	040002	1.	.11111110	.02083
2	040002	040013	.	.00000000	.02083
3	040013	040024	.	.00000000	.02083
4	040024	040035	.	.00000000	.02083
5	040035	040046	.	.00000000	.02083
6	040046	040057	.	.00000000	.02083
7	040057	040070	.	.00000000	.02083
8	040070	040101	1.	.11111110	.04167
9	040101	040112	.	.00000000	.04167
10	040112	040123	.	.00000000	.04167
11	040123	040134	.	.00000000	.04167
12	040134	040145	.	.00000000	.04167
13	040145	040156	3.	.33333331	.10417
14	040156	040167	.	.00000000	.10417
15	040167	040200	.	.00000000	.10417
16	040200	040211	.	.00000000	.10417
17	040211	040222	.	.00000000	.10417
18	040222	040233	.	.00000000	.10417
19	040233	040244	1.	.11111110	.12500
20	040244	040255	4.	.44444442	.20833
21	040255	040266	.	.00000000	.20833
22	040266	040277	4.	.44444442	.29167
23	040277	040310	4.	.44444442	.37500
24	040310	040321	1.	.11111110	.39583
25	040321	040332	6.	.66666663	.52083

TABLE VI (cont.)

26	040332 040343	9.	1.00000000	.70833
27	040343 040354	4.	.44444442	.79167
28	040354 040365	.	.00000000	.79167
29	040365 040376	.	.00000000	.79167
30	040376 040407	.	.00000000	.79167
31	040407 040420	.	.00000000	.79167
32	040420 040431	.	.00000000	.79167
33	040431 040442	.	.00000000	.79167
34	040442 040453	.	.00000000	.79167
35	040453 040464	1.	.11111110	.81250
36	040464 040475	.	.00000000	.81250
37	040475 040506	.	.00000000	.81250
38	040506 040517	1.	.11111110	.83333
39	040517 040530	.	.00000000	.83333
40	040530 040541	1.	.11111110	.85417
41	040541 040552	.	.00000000	.85417
42	040552 040563	1.	.11111110	.87500
43	040563 040574	.	.00000000	.87500
44	040574 040605	.	.00000000	.87500
45	040605 040616	.	.00000000	.87500
46	040616 040627	.	.00000000	.87500
47	040627 040640	.	.00000000	.87500
48	040640 040651	.	.00000000	.87500
49	040651 040662	.	.00000000	.87500
50	040662 040673	.	.00000000	.87500
51	040673 040704	.	.00000000	.87500
52	040704 077777	6.	.66666663	1.00000

TABLE VII

Program Activity Profile for CALC
From 040142 to 040651 in Increments of 7

INTERVAL NO.	FROM	TO	NO. OF HITS	NORMALIZED HITS	NORMAL ACCUM
1	000000	040142	1.	.14285713	.02174
2	040142	040151	.	.00000000	.02174
3	040151	040160	2.	.28571427	.06522
4	040160	040167	.	.00000000	.06522
5	040167	040176	2.	.28571427	.10870
6	040176	040205	.	.00000000	.10870
7	040205	040214	.	.00000000	.10870
8	040214	040223	.	.00000000	.10870
9	040223	040232	4.	.57142854	.19565
10	040232	040241	3.	.42857140	.26087
11	040241	040250	1.	.14285713	.28261
12	040250	040257	.	.00000000	.28261
13	040257	040266	.	.00000000	.28261
14	040266	040275	2.	.28571427	.32609
15	040275	040304	3.	.42857140	.39130
16	040304	040313	3.	.42857140	.45652
17	040313	040322	1.	.14285713	.47826
18	040322	040331	7.	1.00000000	.63043
19	040331	040340	5.	.71428573	.73913
20	040340	040347	2.	.28571427	.78261
21	040347	040356	.	.00000000	.78261
22	040356	040365	1.	.14285713	.80435
23	040365	040374	.	.00000000	.80435
24	040374	040403	.	.00000000	.80435
25	040403	040412	.	.00000000	.80435

TABLE VII (cont.)

26	040412	040421	.	.00000000	.80435
27	040421	040430	.	.00000000	.80435
28	040430	040437	.	.00000000	.80435
29	040437	040446	.	.00000000	.80435
30	040446	040455	1.	.14285713	.82609
31	040455	040464	.	.00000000	.82609
32	040464	040473	.	.00000000	.82609
33	040473	040502	.	.00000000	.82609
34	040502	040511	1.	.14285713	.84783
35	040511	040520	1.	.14285713	.86957
36	040520	040527	.	.00000000	.86957
37	040527	040536	.	.00000000	.86957
38	040536	040545	.	.00000000	.86957
39	040545	040554	.	.00000000	.86957
40	040554	040563	1.	.14285713	.89130
41	040563	040572	.	.00000000	.89130
42	040572	040601	.	.00000000	.89130
43	040601	040610	.	.00000000	.89130
44	040610	040617	.	.00000000	.89130
45	040617	040626	.	.00000000	.89130
46	040626	040635	.	.00000000	.89130
47	040635	040644	.	.00000000	.89130
48	040644	040653	.	.00000000	.89130
49	040653	040662	.	.00000000	.89130
50	040662	040671	.	.00000000	.89130
51	040671	040700	.	.00000000	.89130
52	040700	077777	5.	.71428573	1.00000

TABLE VIII

Program Activity Profile for DO Loop of CALC
From 040214 to 040365 in Increments of 3

INTERVAL NO.	FROM	TO	NO. OF HITS	NORMALIZED HITS	NORMAL ACCUM
1	000000	040214	4.	.66666663	.08696
2	040214	040217	.	.00000000	.08696
3	040217	040222	4.	.66666663	.17391
4	040222	040225	.	.00000000	.17391
5	040225	040230	.	.00000000	.17391
6	040230	040233	.	.00000000	.17391
7	040233	040236	2.	.33333331	.21739
8	040236	040241	.	.00000000	.21739
9	040241	040244	.	.00000000	.21739
10	040244	040247	4.	.66666663	.30435
11	040247	040252	2.	.33333331	.34783
12	040252	040255	.	.00000000	.34783
13	040255	040260	.	.00000000	.34783
14	040260	040263	.	.00000000	.34783
15	040263	040266	.	.00000000	.34783
16	040266	040271	.	.00000000	.34783
17	040271	040274	.	.00000000	.34783
18	040274	040277	1.	.16666666	.36957
19	040277	040302	1.	.16666666	.39130
20	040302	040305	.	.00000000	.39130
21	040305	040310	3.	.50000000	.45652
22	040310	040313	.	.00000000	.45652
23	040313	040316	3.	.50000000	.52174
24	040316	040321	.	.00000000	.52174
25	040321	040324	.	.00000000	.52174

TABLE VIII (cont.)

26	040324 040327	2.	.33333331	.56522
27	040327 040332	.	.00000000	.56522
28	040332 040335	6.	1.00000000	.69565
29	040335 040340	.	.00000000	.69565
30	040340 040343	1.	.16666666	.71739
31	040343 040346	2.	.33333331	.76087
32	040346 040351	.	.00000000	.76087
33	040351 040354	.	.00000000	.76087
34	040354 040357	.	.00000000	.76087
35	040357 040362	1.	.16666666	.78261
36	040362 040365	.	.00000000	.78261
37	040365 040370	.	.00000000	.78261
38	040370 040373	.	.00000000	.78261
39	040373 040376	.	.00000000	.78261
40	040376 040401	.	.00000000	.78261
41	040401 040404	.	.00000000	.78261
42	040404 040407	.	.00000000	.78261
43	040407 040412	.	.00000000	.78261
44	040412 040415	.	.00000000	.78261
45	040415 040420	.	.00000000	.78261
46	040420 040423	.	.00000000	.78261
47	040423 040426	.	.00000000	.78261
48	040426 040431	.	.00000000	.78261
49	040431 040434	.	.00000000	.78261
50	040434 040437	.	.00000000	.78261
51	040437 040442	.	.00000000	.78261
52	040442 077777	10.	1.66666675	1.00000

TABLE IX

Load Map for CDRVR and CALC

```
CDRVR   40002 40141
CALC    40142 40651

ERRO    40652 40760   750701   24998-16001
SQRT    40761 41107   751101   24998-16001
.OPSY   41110 41147   750701   24998-16001
CLRIO   41150 41156   750701   24998-16001
..FCM   41157 41173   750701   24998-16001
REIO    41174 41276   92001-16005 741120
ERO.E   41277 41277   750701   24998-16001
.PWR2   41300 41332   750701   24998-16001
.DFER   41333 41404   750701   24998-16001

ENTRY POINTS

 *CDRVR   40076
 *EXEC    12446
 *CLRIO   41150
 *CALC    40147
 *.FMP   105040
 *.FDV   105060
 *.FAD   105000
 *.FSB   105020
 *..FCM   41157
 *.MPY   100200
 *.DLD   104200
 *.DST   104400
 *.ENTR   37201
 *SQRT    40771
 *FLOAT  105120
 *ERRO    40652
 *REIO    41200
 *ERO.E   41277
 *.OPSY   41110
 *.PWR2   41300
 *.ZRNT   02001
 *.ZPRV   02001
 *.DFER   41333
 *$LIBR   12665
 *$LIBX   13463
```

this interval immediately follows a floating point divide instruction, which is the slowest of all the floating point instructions. This accounts for the high concentration at that point.

Close examination of the CALC listing shows that all the "hits" in the specified range in Table VIII occur in the "DO" loop of the program. Furthermore, the "hits" are concentrated in the area of the loop where many floating point operations take place. The microprogramming effort in CALC should be concentrated on this loop, microcoding the entire loop if possible.

Summary

This chapter covered the analysis techniques for finding the time-consuming areas of a higher level or assembly language program. Two programs were analyzed in detail using an activity profile generator program. The first program, the wind tunnel stress routine, showed a high activity concentration in a very small loop. The second program, the laser materials modeling routine, had its area of high activity also in a loop. The loop of the second program, however, was much larger, containing several lengthy FORTRAN statements. Both of the loops of the two programs could be microcoded.

IV. Requirements, Design, Implementation and Test of a Microprogram for the Wind Tunnel Control Program

Introduction

The previous chapter covered the analysis of the stress calculation routine (STRES) for the wind tunnel control program. This analysis showed that about 80 per cent of the program activity in the assembly language subroutine of STRES called SPEED occurred in one loop segment of SPEED. This chapter covers the detailed requirements, design, implementation and test of a microprogram called LOADS to replace this loop segment in SPEED.

LOADS Requirements

LOADS is the microprogram designed to replace the loop segment labeled LOOP2 in SPEED. LOOP1 is actually the loop in which most of the activity was found to occur, but it is nested within LOOP2. Because of this relationship between the two loops, it was decided to include LOOP2 in the design of LOADS.

First and second level data flow diagrams (DFDs) (Ref. 30:Chapt. 4) of SPEED are given in Figure 4 to show the relationship of LOOP2 to the rest of SPEED. As shown in the diagrams, SPEED performs three major functions -- computation of loads (forces), moments, and stresses. LOOP2, along with its inner loop labeled LOOP1, performs the load

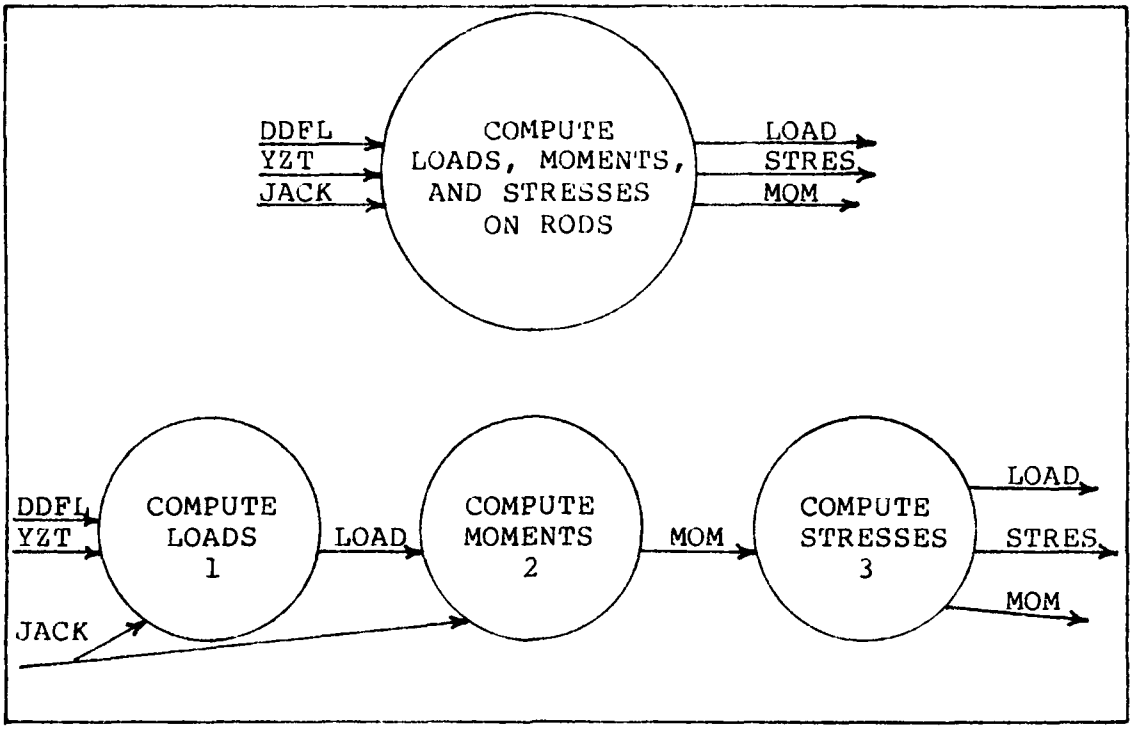


Figure 4. Level 1 and 2 DFDs for Subroutine SPEED

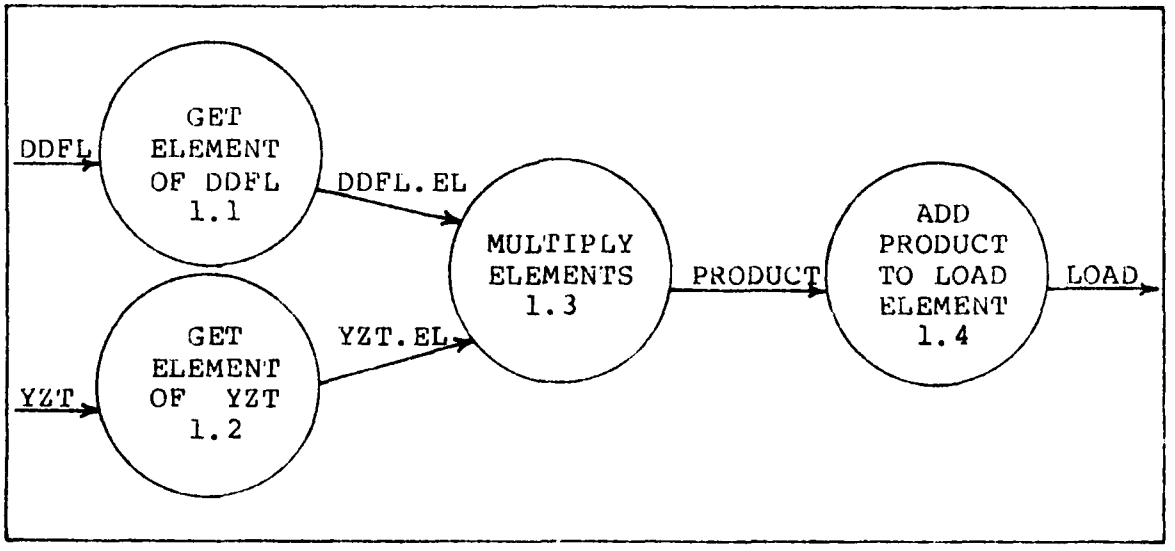


Figure 5. DFD for LOADS Microprogram

computation function. The purpose of this function is to calculate the loads on the individual flexible rods of the wind tunnel. Calculation of the loads is an interim calculation to computing the moments and stresses on each rod, as shown in the second level DFD of Figure 4. The calculation of the loads consists of a simple matrix multiplication.

Matrix multiplication is defined as follows (Ref. 31:343): Given $A=(a_{ij})$, an m x n matrix and $B=(b_{ij})$, an n x p matrix. Then the product AB is a matrix $C=(c_{ij})$ where:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{ik}$$

and the matrix C is of order m x p. The two matrices involved in the load calculation are called DDFL and YZT.

DDFL is a 13 by 13 matrix, which represents the inverse matrix of the deflections of a single wind tunnel rod due to a unit load applied to the rod at one point. There are 13 jacks attached to each rod, giving 13 deflection points and 13 points to apply a unit load.

YZT is a 14 by 1 matrix which represents the deflections in inches from the neutral position of the 13 jacks attached to each rod. YZT(1) represents the point at which a rod is attached to the tunnel wall, and thus always has a deflection of zero. YZT(2) through YZT(4) are the deflections of the manual jacks used to position each rod.

YZT(5) through YZT(14) are the deflections of the ten electric jacks used for the same purpose. The values for the electric jack deflections are actually the periodic readings from potentiometers attached to each rod (one potentiometer per rod). The first element of YZT, YZT(1), is not used in the matrix multiplaction. This makes the YZT matrix effectively a 13 by 1 matrix, satisfying the dimension requirements for matrix multiplication.

The result of the multiplication of DDFL and YZT is a 13 by 1 matrix called LOAD. LOAD is actually a 14 by 1 matrix like YZT with the first element set to zero, and the remaining 13 elements are the result of the matrix multiplication. The reason the YZT and LOAD matrices have an extra element is because of requirements in other routines of the wind tunnel control program. For the purpose of the matrix multiplication, they are 13 by 1 matrices and will be referred to as such.

The data flow for the matrix multiplication of DDFL and YZT is shown in the DFD of Figure 5. This DFD is a further breakdown of the "COMPUTE LOADS" "bubble" of the DFD of Figure 4. The data flow here is very simple. Elements of DDFL and YZT are obtained from their respective matrices. The two elements are multiplied, and the product is added to the appropriate LOAD element. The control involved in this process cannot be shown in a DFD, but is described in the following structured English (Ref. 30:Chapt. 6) requirements specification of the DDFL and YZT

matrix multiplication routine:

```
REPEAT UNTIL NO MORE ROWS IN THE DDFL MATRIX
     SET NEXT ELEMENT OF LOAD MATRIX TO 0
     REPEAT UNTIL NO MORE COLUMNS IN THE DDFL
     MATRIX
          MULTIPLY NEXT DDFL AND YZT ELEMENTS
          ADD THE PRODUCT TO THE CURRENT LOAD
          ELEMENT
          POINT TO THE NEXT YZT ELEMENT
          POINT TO THE NEXT DDFL COLUMN
     END
     POINT TO THE NEXT LOAD ELEMENT
     POINT TO THE NEXT DDFL ROW
END
```

The matrix multiplication consists of two loops, one within the other, as shown in the above structured English specification. The inner loop corresponds to LOOP1 in SPEED, and the outer loop corresponds to LOOP2.

Design of LOADS

The basic design of the LOADS microprogram is shown in Figure 6 in the form of a structure chart (Ref. 30:Chapt. 7). This chart is the result of transform analysis (Ref. 30:Chapt. 9), which is a design technique that builds a system around the concept of data transformation. In the case of LOADS, the data elements of the DDFL and YZT matrices are transformed into an element of the LOAD matrix. This transformation is shown in the DFD of LOOP1, from which the structure chart is drawn. The reader should note that the data names on the structure chart are for design purposes only and do not actually exist in the microcode. Data at the micro-level exists in registers, and the ap-

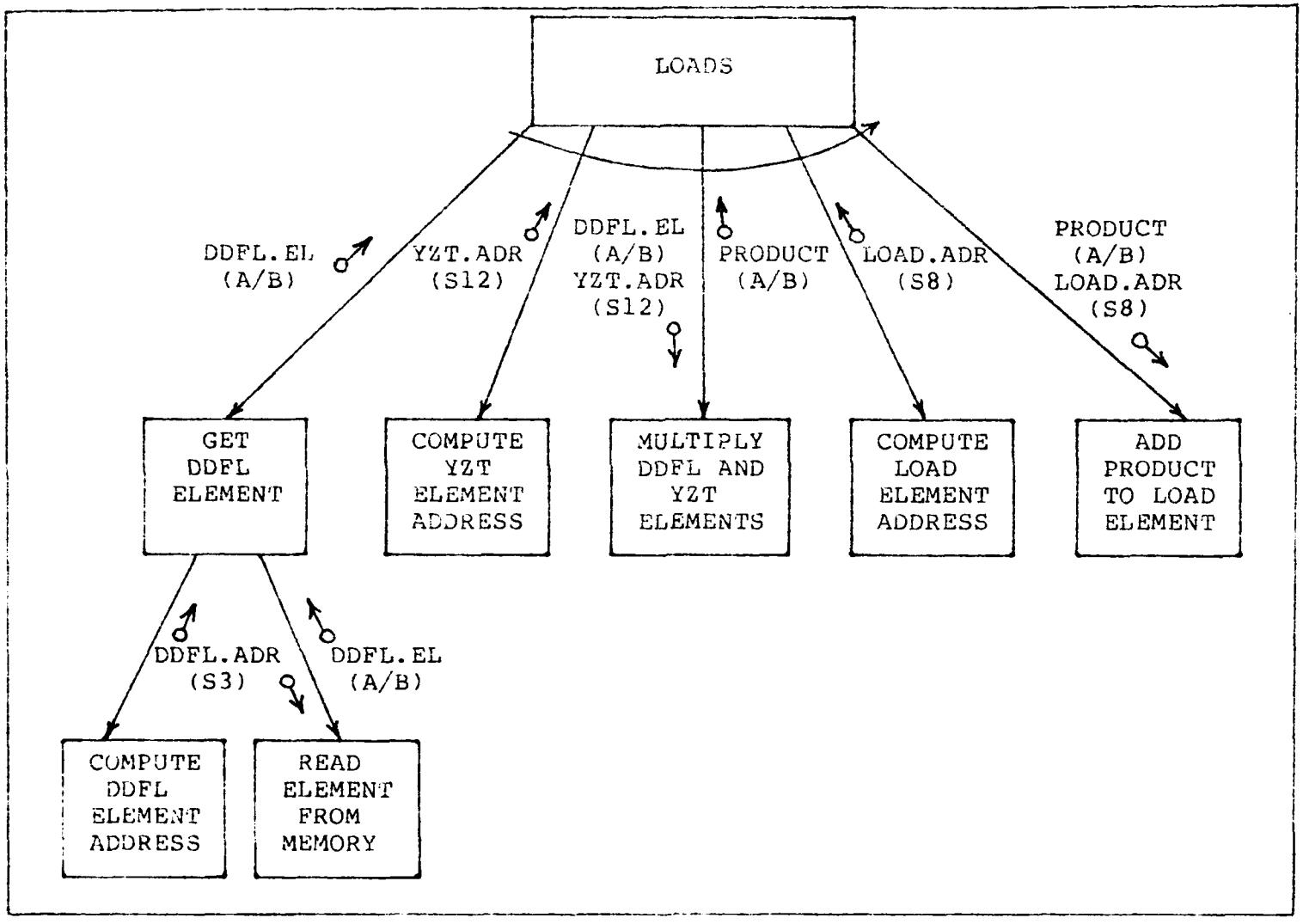


Figure 6. LOADS Microprogram Structure Chart

plicable registers used for temporary storage are shown in parenthesis below the corresponding data name.

If LOADS was to be implemented in a higher level language, the design of the routine would essentially be done. It is a simple process to code a FORTRAN or PASCAL program from the above structured English using the modular design of the structure chart. Implementing the routine in microcode, however, requires the design to go to an even lower level. The following algorithmic steps take the design to a sufficiently low level to write the microcode:

1. Read calling parameters -- addresses for DDFL, YZT, and LOAD matrices -- from memory and store into their respective scratch registers.
2. Store an outer loop count of 13 into a loop counter register.
3. Store an inner loop count of 13 into a loop counter register.
4. Set the current LOAD matrix element to zero.
5. Read the current DDFL matrix element from memory into the A/B registers.
6. Read the current YZT matrix element from memory.
7. Call the floating point multiply routine.
8. Read the current LOAD matrix element from memory.
9. Call the floating point add routine.
10. Store the result into the LOAD matrix element.
11. Increment the DDFL address register.
12. Increment the YZT address register.
13. Decrement the inner loop counter register.
14. If the counter does not equal zero, go back to step 5.
15. Increment the LOAD address register.
16. Decrement the outer loop counter register.
17. If the counter does not equal zero, go back to step 4.
18. Return to the calling assembly language routine.

Implementation of LOADS

The process of implementing the design of LOADS in microcode is straightforward. Use of the HP microassembler

(Ref. 32:5-1) makes the coding very similar to coding in assembly language. The resultant microprogram is listed in Appendix G along with the modified version of SPEED called MSPED, required to invoke the microprogram.

Some of the limitations of the HP architecture have a significant effect on the microcode. Because only one register is available for subroutine return addresses in the HP 21MX M-Series, subroutine calls cannot be made from other subroutines without losing the original return address. This is a significant problem when using control store ROM routines such as the floating point multiply and add routines required by LOADS. These routines call other ROM routines, so they cannot be used directly. One way around this problem is to duplicate the routines in WCS and "jump" directly to and from these routines. Duplicating the routines is no problem as all the ROM routines are documented (Ref. 32:Appendix E), but they do use much valuable WCS space.

Another problem with using the ROM routines is that they use many of the available scratch registers. For example, the floating add and multiply routines and their associated subroutines use ten of the twelve available scratch registers. Scratch registers are very important since data cannot be stored in WCS in the HP 21MX. With the routines in WCS, the register usage can be reorganized somewhat. Doing this in LOADS freed two more registers.

Testing

The plan for testing the LOADS microprogram consisted of two major phases -- a module test and a system test. The module test was conducted on the AFIT HP system using the special driver program SDRVR to drive LOADS (via STRES and SPEED). The purpose of this test phase was to show that LOADS would produce the same output as the assembly language code segment replaced by LOADS. The system test was run on the AFWAL wind tunnel control computer. The purpose of this test phase was to show that the LOADS microprogram would load and execute correctly on the system for which it was designed.

Module Test. The module test plan consisted of two parts: (1) verification of the program output, and (2) determination of the speed improvement of the microprogram. Imbedded in the data and assignment statements of SDRVR were known inputs for the subroutine STRES, which would produce known stress and moment calculation outputs. The goal of this part of the module test was to duplicate those outputs using the microprogrammed version of the program. HP's Micro Debug Editor (MDE) (Ref. 32:5-21) was used for loading the LOADS microprogram into WCS, and for debugging the microprogram. Debugging consisted of setting breakpoints within the microprogram and analyzing register contents when the breakpoints were reached.

Determination of the speed improvement of STRES and SPEED through the use of the LOADS microprogram was accom-

plished through executive calls to read the system clock. It was necessary to call STRES (and therefore SPEED) 100 times from a loop in order to get accurate measurements. The timing tests showed that the microprogrammed version of SPEED (using LOADS) was 36 per cent faster than the original version. Since the loop replaced by LOADS represented 80 per cent of the total execution time of SPEED, LOADS was actually 45 per cent faster than the loop it replaced. The speed increase in SPEED made its calling program STRES 31 per cent faster.

The 45 per cent speed increase (almost two times as fast) is somewhat less than the gains of six to ten times (Ref. 5:98) or two to twenty times (Ref. 9:49) reported in the literature. Close analysis of the assembly language for the loop explains the difference. Totaling the instruction times for floating point and non-floating point instructions shows that 46 per cent of the loop's execution time is spent in the two floating point instructions. Since the microcoded version of the program uses these same floating point routines, no speed improvement can be made to 46 per cent of the loop, and the best possible speed improvement to the loop is 54 per cent. A 45 per cent speed increase is therefore, not only reasonable, but quite good.

No major problems were encountered in the module test, although several small problems were encountered. One problem was a logic error in the loop structure of LOADS,

wnich made it attempt to write to main memory beyond the bounds of the driver program. This situation caused a system memory protect error detected by the special memory protect hardware on the AFIT system. This optional hardware feature showed its usefulness in protecting memory from an untested microprogram. As one of the HP manuals warns, "execution of an unproven microprogram can have unpredictable and undesirable results, including the destruction of the system" (Ref. 32:5-16). Once the logic error of LOADS was corrected, the program produced the correct output, and the module test was successful.

System Test. The system test of LOADS consisted of the same two steps as the module test -- output verification and speed improvement. In this test, however, the inputs to the program were from the operational wind tunnel control system hardware, and the microprogram was driven by the operational software. Also, the speed improvement measurement here was concerned with the number of additional rods of the wind tunnel that could be driven.

Two problems were encountered in the system test. One problem was loading the microprogram into the WCS of the wind tunnel computer, and the other was executing the microprogram after it was loaded.

Loading the microprogram was a problem because of the difference in operating systems of the AFIT and wind tunnel machines. The AFIT system uses a real-time executive system, RTE-III, and the wind tunnel system uses an older disk

operating system, DOS-III. Microprogramming support software was available for the DOS-III system, but had not been procurred for the wind tunnel machine. The problem was solved by writing a special WCS loader program in assembly language using the I/O instruction sequences given in the WCS manual (Ref. 33:3-1). The listing for this program called WCSLD is in Appendix H. Initial attempts to run WCSLD on the wind tunnel machine failed. The problem was traced to a bad WCS board, and the microprogram was successfully loaded to a new board. WCSLD will not run on an RTE system with the memory protect option installed because of the direct I/O instructions used.

The next problem occurred in executing the LOADS microprogram after it had been loaded. The program seemed to work, but zero values were returned for the stress calculations. This indicated that the microprogram was not being invoked by the assembly language instruction in SPEED. This problem was traced to an improper combination of address jumper wires on the WCS board. Removal of two jumper wires fixed this problem, and the program ran successfully.

The speed improvement measurement was made by interactively increasing the number of concurrent rod adjustments, and monitoring the adjustment process on a special light panel. The light panel readily indicated when the program bogged down because of too many concurrent adjustments. The test showed that four rods could be adjusted reliably using the microprogrammed version of the program.

The old program could only handle three reliably. It was felt that the microprogrammed version was probably close to five rods, but this could not be validated.

Summary

This chapter covered the requirements, design, implementation, and testing of a microprogram called LOADS, designed for use in the wind tunnel control program. The application of the microprogram showed a 31 per cent speed improvement in the stress calculation routine of the control program. This improvement resulted in a 33 per cent operational improvement of the rod adjustment process of the tunnel, by increasing the capability from three to four concurrent rod adjustments. The loading and execution of LOADS on a DOS-III system showed that microprograms can be run on this system without microprogramming software support.

V. Requirements, Design, Implementation and Test of a Microprogram for the Laser Materials Model Program

Introduction

As discussed in Chapter II, the laser materials modeling program is used to calculate the real and imaginary parts of refractive index, an important measure of laser materials. Chapter III covered the analysis of the routine called CALC which performs this refractive index calculation. The analysis showed that microprogramming could be applied to the one loop in CALC. This chapter covers the requirements, design, implementation and test of a microprogram called MCALC to replace this loop in CALC.

MCALC Requirements

The real and imaginary parts of the refractive index are given by the following equations:

$$n^2-k^2=\varepsilon_0 \sum_i 4\pi\rho_i\nu_i^2 \frac{\nu_i^2-\nu^2}{(\nu_i^2-\nu^2)^2+\gamma_i^2\nu_i^2\nu^2} - x\frac{\nu_p^2}{\nu^2+\nu_r^2}$$

$$nk= \sum_i 2\pi\rho_i\nu_i^2 \frac{\gamma_i\nu_i\nu}{(\nu_i^2-\nu^2)+\gamma_i^2\nu_i^2\nu^2} + x\frac{\nu_r\nu_p^2}{\nu(\nu^2+\nu_i^2)}$$

These are the same two equations which were given in the CALC requirements in Chapter II. The reader may wish to refer back to that chapter for parameter descriptions and

dimensions, although they are not needed for the discussion here. CALC receives as inputs all the parameters required to evaluate the two equations. Once evaluated, the equations can be solved simultaneously for n and k, the real and imaginary parts of refractive index.

The data flow for CALC is shown in the level 1,2, and 3 DFDs in Figure 7. Data flow name definitions are given in Table X. The first level gives the overall function of CALC -- to compute n and k. Level 2 divides this function into the two major tasks of (1) evaluating the two equations and (2) solving these two equations simultaneously. Level 3 further divides the first task into four subtasks. The second task of solving the equations is of no further interest here, since this task is not to be microcoded, and a level 3 diagram is not given.

The level 3 diagram of the equation evaluation task, however, is of particular interest since it provides the basis for the MCALC microprogram. As noted before, this diagram divides the equation evaluation task into four subtasks. The requirement for MCALC is to perform the summation part of the equation evaluation as shown in "bubble" 1.2. This "bubble" then becomes the first level of the MCALC DFD shown in Figures 8 and 9.

The first level of the MCALC DFD shows three inputs -- F, B, and F2. These inputs consist of all the equation parameters required for the evaluation of the summation parts of the two equations. One additional input to MCALC

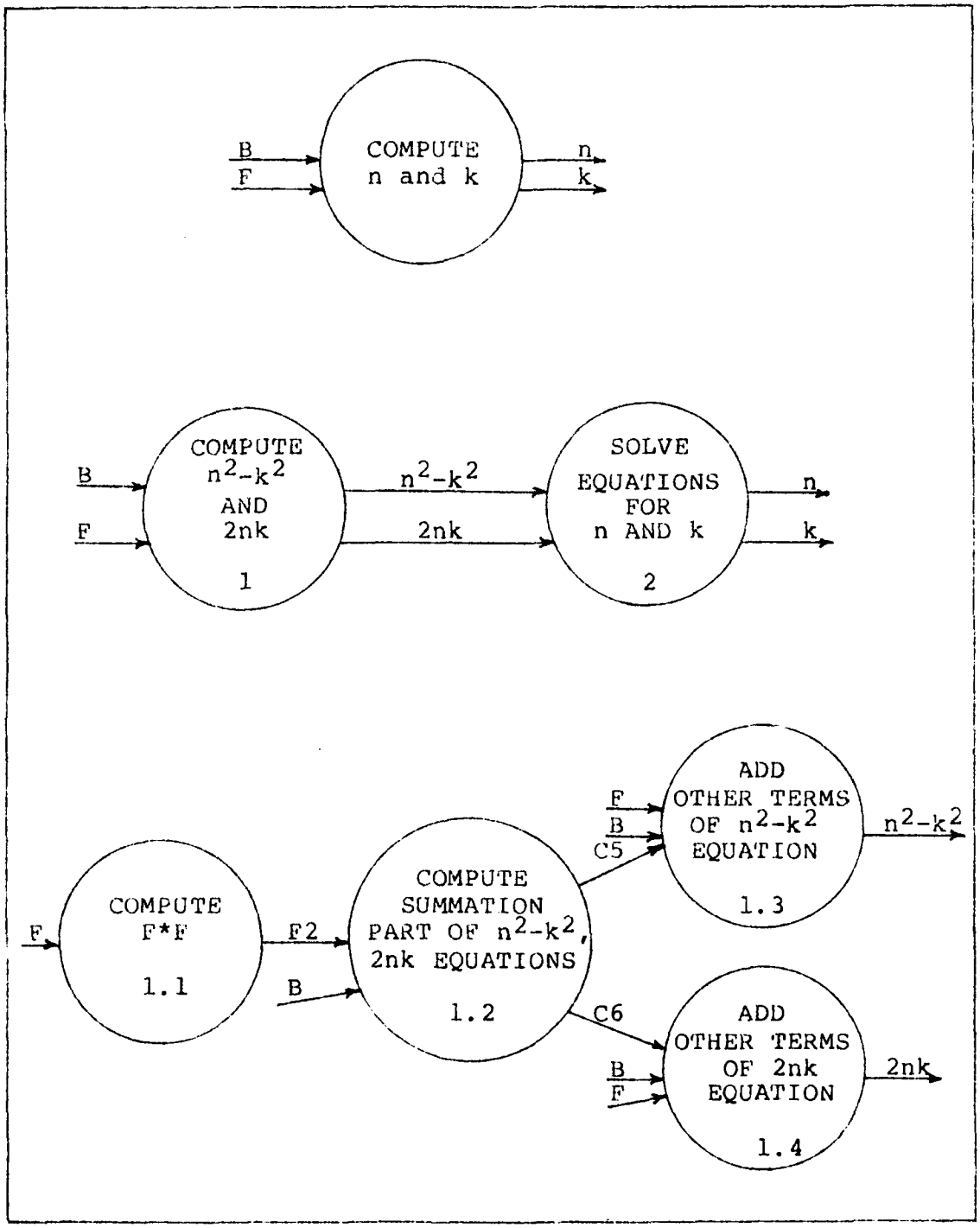


Figure 7. Level 1,2, and 3 DFDs for Subroutine CALC

TABLE X
Data Flow Name Definitions

Data Flow Name	Description	Alias or Equivalent	Equation Symbol
B	Array (30 X 1) containing equation parameter values input from a peripheral poteniometer board.	None	None
GAMMA.NU	Product of damping factor and frequency of resonance	B(1),B(4), B(7),B(10), B(13),B(16), B(19),B(22)	$\gamma_i \nu_i$
NU	Frequency of resonance	B(2),B(5), B(8),B(11), B(14),B(17), B(20),B(23)	ν_i
RHO	Strength of resonance	B(3),B(6), B(9),B(12), B(15),B(18), B(21),B(24)	σ_i
n	Real part of refractive index	None	n
k	Imaginary part of refractive index	None	k
F	Radiation frequency	None	ν
F2	Radiation freq squared	F*F	ν^2

TABLE X (Cont.)
Data Flow Name Definitions

Data Flow Name	Description	Alias or Equivalent	Equation Symbol
C1	Intermediate calculation	GAMMA.NU*F	Combination
C2	Intermediate calculation	NU*NU	of
C3	Intermediate calculation	C2 F2	Previously
C4	Intermediate calculation	RHO*C2/ C1*C1+C3*C3	Defined
C5	Intermediate calculation	C5+C3*C4	Symbols
C6	Intermediate calculation	C6+C1*C4	
C1.C1	Intermediate calculation	C1*C1	
C3.C2	Intermediate calculation	C3*C3	
C1.C3	Intermediate calculation	C1*C1+C3*C3	
RHO.C2	Intermediate calculation	RHO*C2	
C3.C4	Intermediate calculation	C3*C4	
C1.C4	Intermediate calculation	C1*C4	

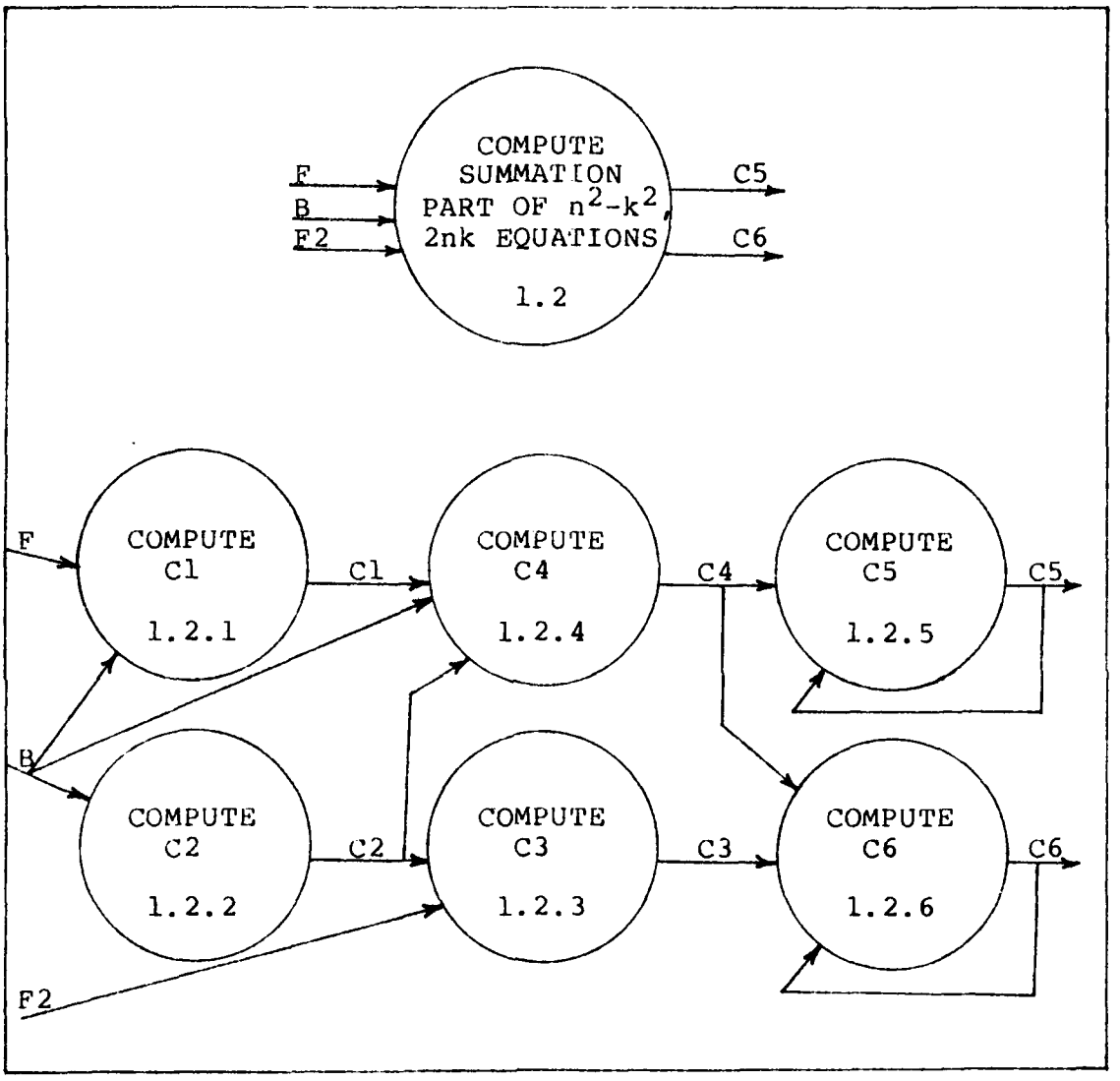


Figure 8. Level 1 and 2 DFDs for MCALC Microprogram (Levels 3 and 4 of CALC)

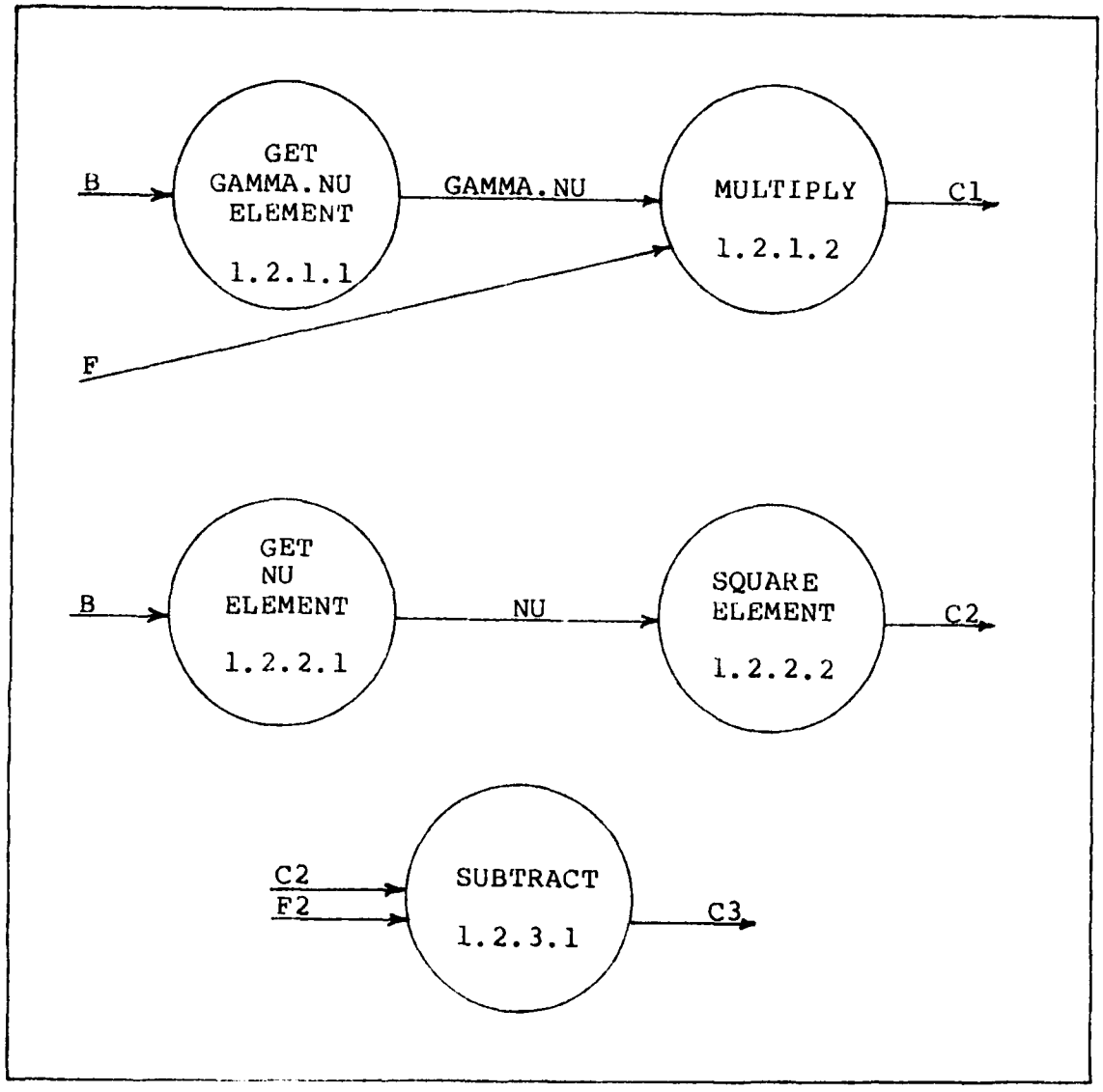


Figure 9. Level 3 DFDs for MCALC Microprogram (Level 5 of CALC)

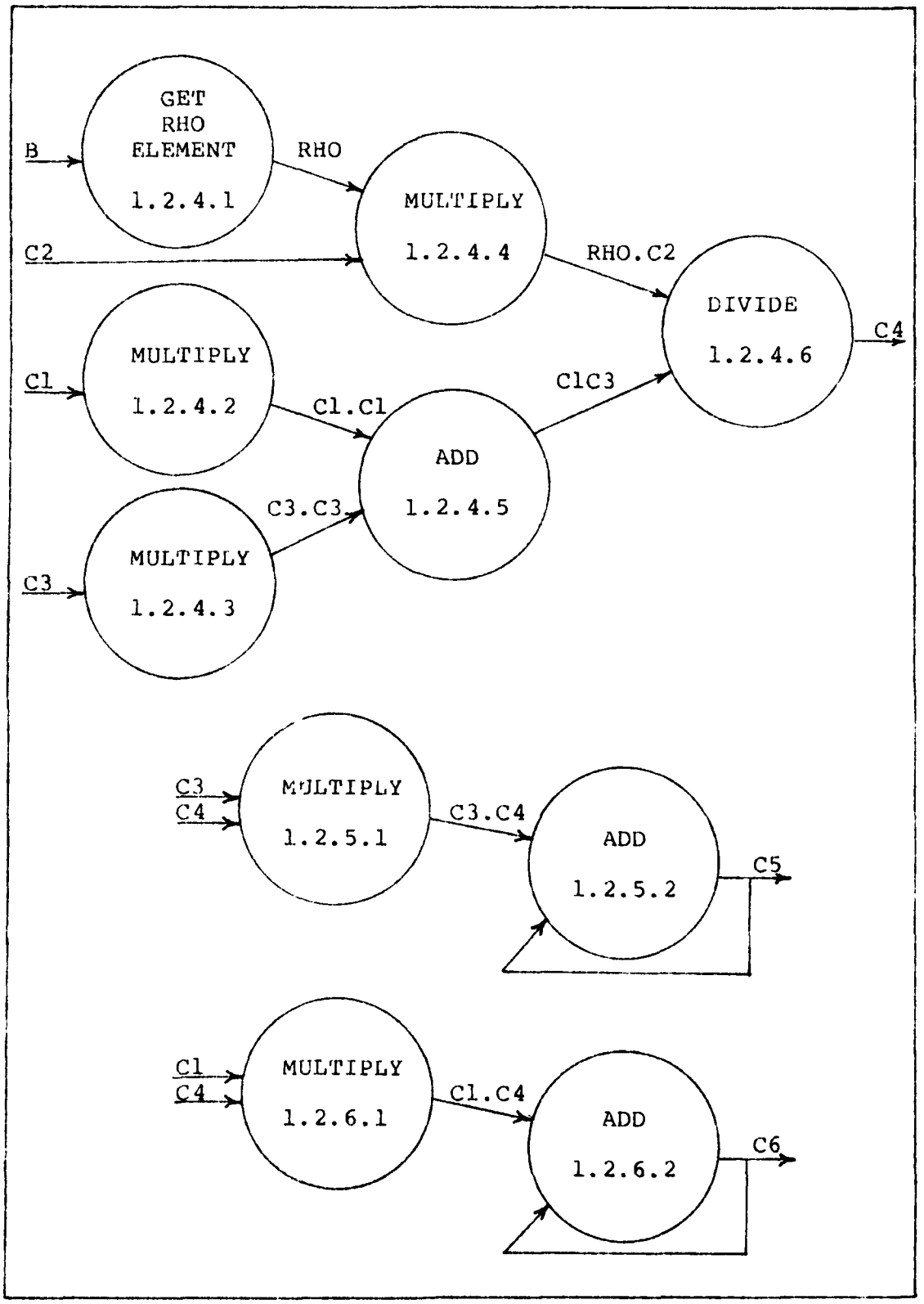


Figure 9. (Cont.)

which cannot be shown on a DFD is the upper limit of the summation. This variable, called JJ in CALC, can have a value between one and eight and serves as the loop count for MCALC. The reason JJ cannot be shown on a DFD is because it is a control variable rather than a data variable. The two outputs of MCALC, C5 and C6, are the evaluated summations in the n^2-k^2 and 2nk equations respectively.

The second level of the DFD divides the summation computation into four intermediate computations, computations C1 through C4, and the two final computations of C5 and C6. All of the variable names used so far in the DFDs are identical to those used in the original FORTRAN loop shown in the listing of CALC given in Appendix I. Descriptions of these variables describing their relationships to the original equations are given in Table X.

The third level of the DFD gives further details of the computations of C1 through C6. Many of the data flow names used here are for the purpose of the DFD only and are not found in CALC or MCALC. These are also described in Table X.

Design of MCALC

The basic design of MCALC is shown in the structure chart of Figure 10. Like the wind tunnel microprogram LOADS, MCALC was designed using transform analysis of the DFDs. The inputs B, F, and F2 are transformed into intermediate calculations C1 through C4. C1, C3, and C4 are

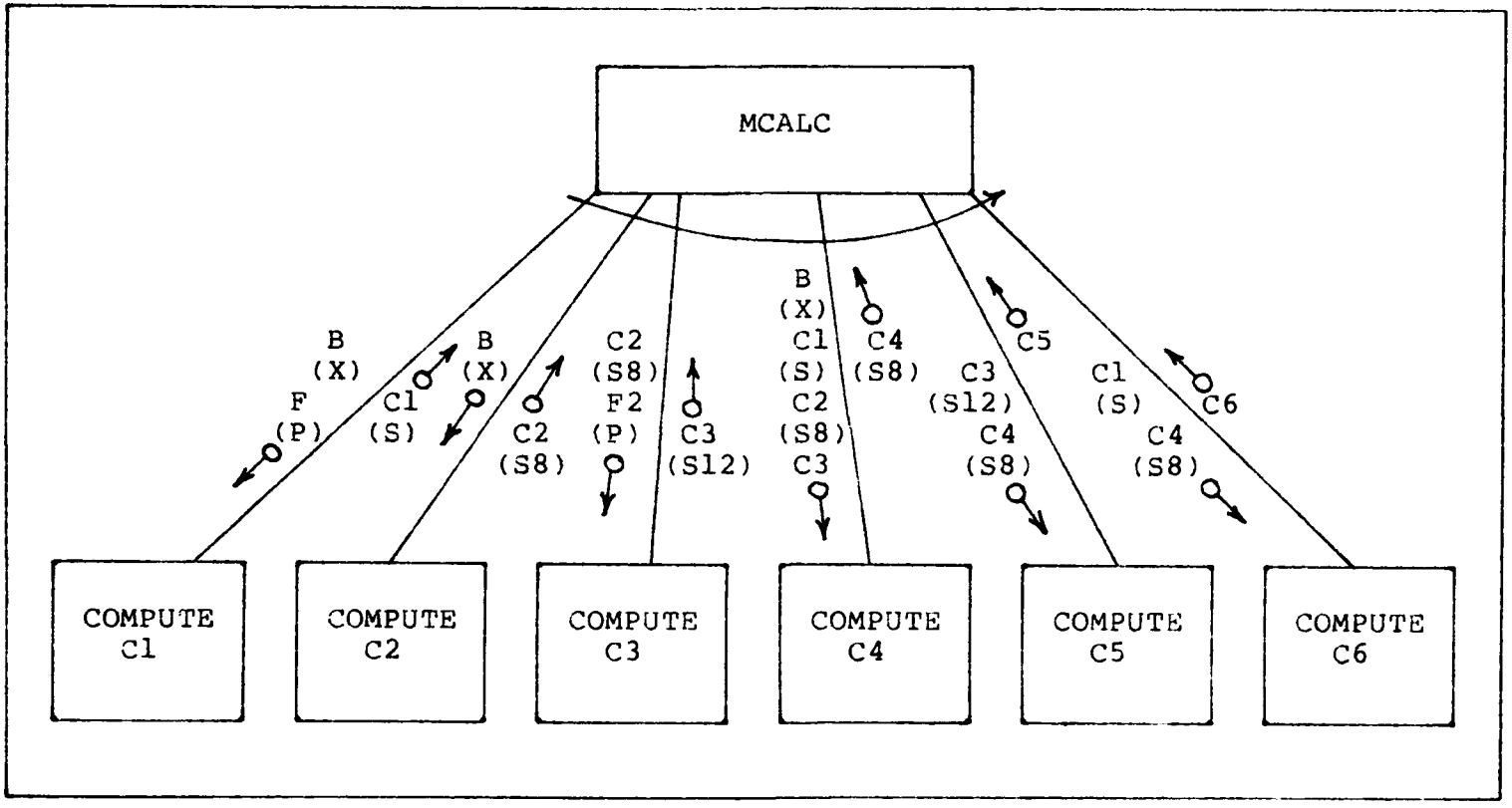


Figure 10. MCALC Microprogram Structure Chart (Levels 1 and 2)

then transformed into C5 and C6, the two outputs of MCALC. Figure 11 consists of six structure charts which are the result of second-level factoring (Ref. 30:180) or further refinement of the structure chart of Figure 10. These six structure charts show the calculations of C1 through C6. As was pointed out in the previous chapter, the variable names used in the structure chart do not actually exist in the microcode, since all data or data addresses reside in registers or main memory. Registers which are associated with the variables are shown in parenthesis below the variable names.

To complete the design of MCALC, detailed algorithmic (register transfer language, pseudo English) steps are written using the structure of the structure charts. These steps are shown below:

1. Initialization
   a. Read calling parameters from registers/memory.
   b. Save parameters in appropriate registers:
      X <-- B address, Y <-- JJ (loop count)
      S <-- TMP1 address, S8 <-- TMP2 address
      S12 <-- TMP3 address
2. Calculate C1. C1=GAMMA.NU(I)*F
   a. Read GAMMA.NU element from B array into A/B.
   b. Get F address from parameter list and put into S3.
   c. Call FMPY (Floating Point Multiply).
   d. Save C1 in TMP1.
3. Calculate C2. C2=NU(I)*NU(I)
   a. Read NU element from B array into A/B.
   b. Put NU address into S3.
   c. Call FMPY.
   d. Save C2 in TMP2.
4. Calculate C3. C3=C2-F2
   a. Get F2 address from parameter list and put into S3.
   b. Call FSUB (Floating Subtract). (C2 still in A/B).
   c. Save C3 in TMP3.
5. Calculate C4. C4=(RHO(I)*C2)/(C3*C3+C1*C1)

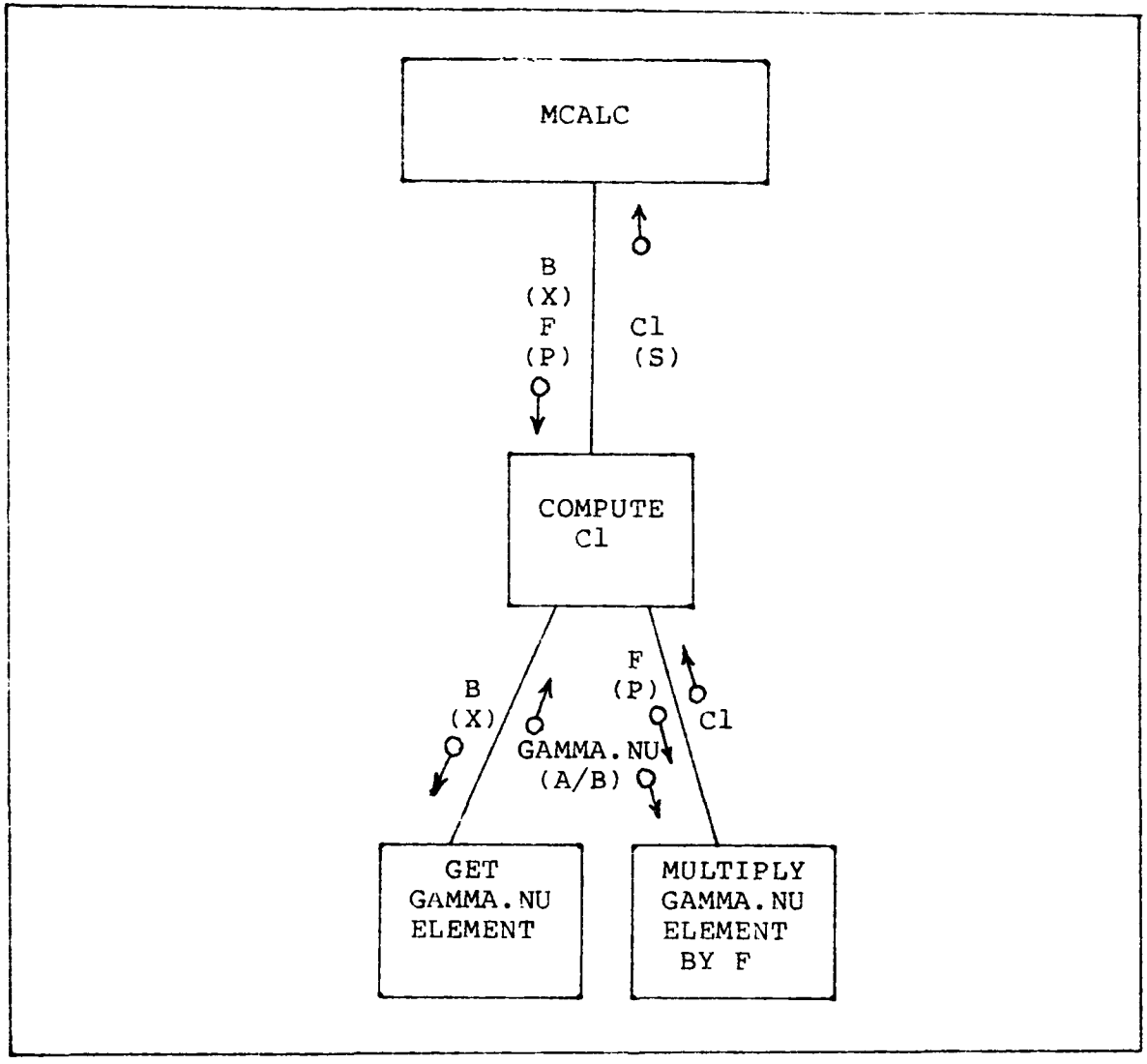


Figure 11. MCALC Microprogram Structure Chart (Level 2 Factor)

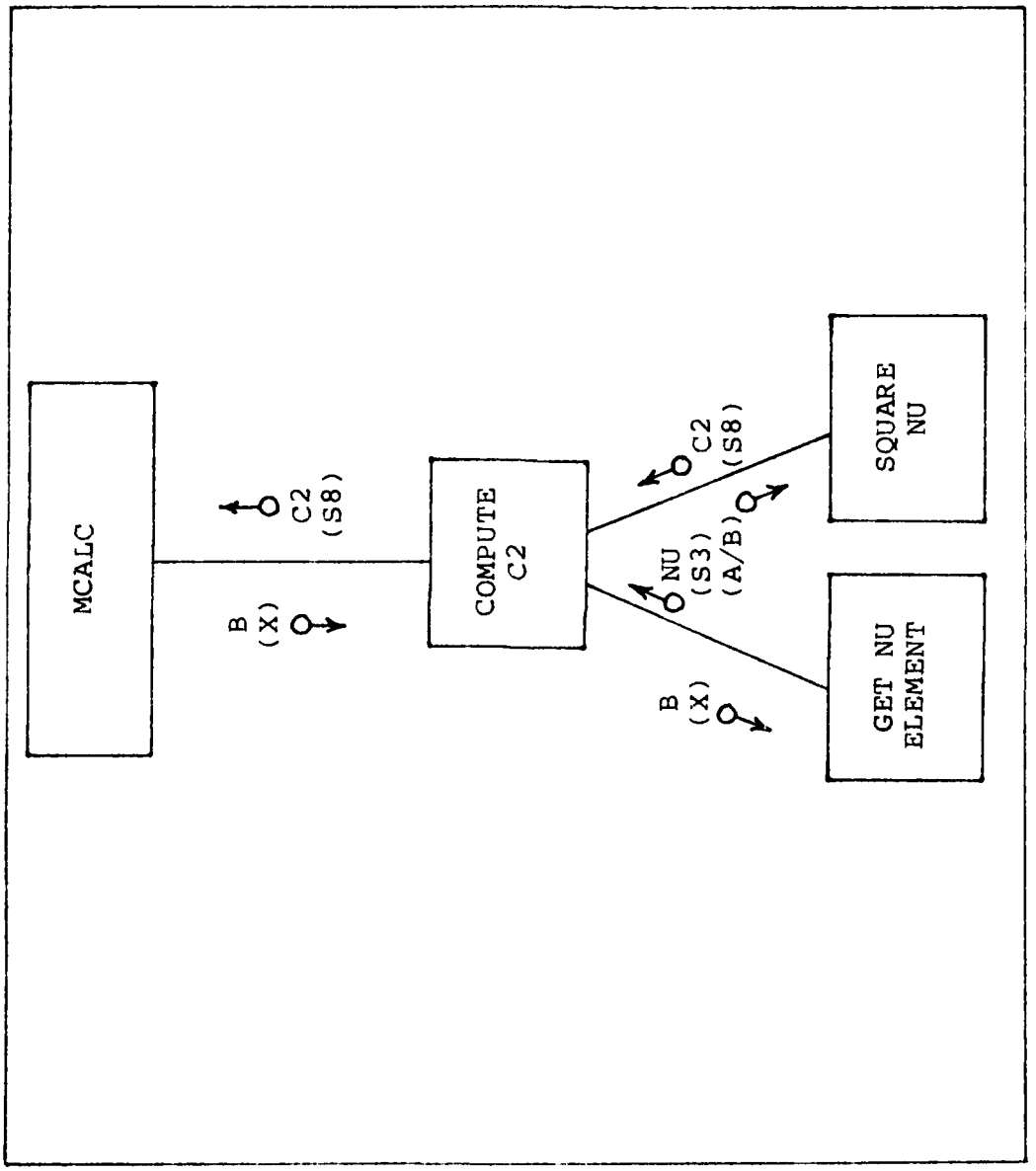


Figure 11. Cont.

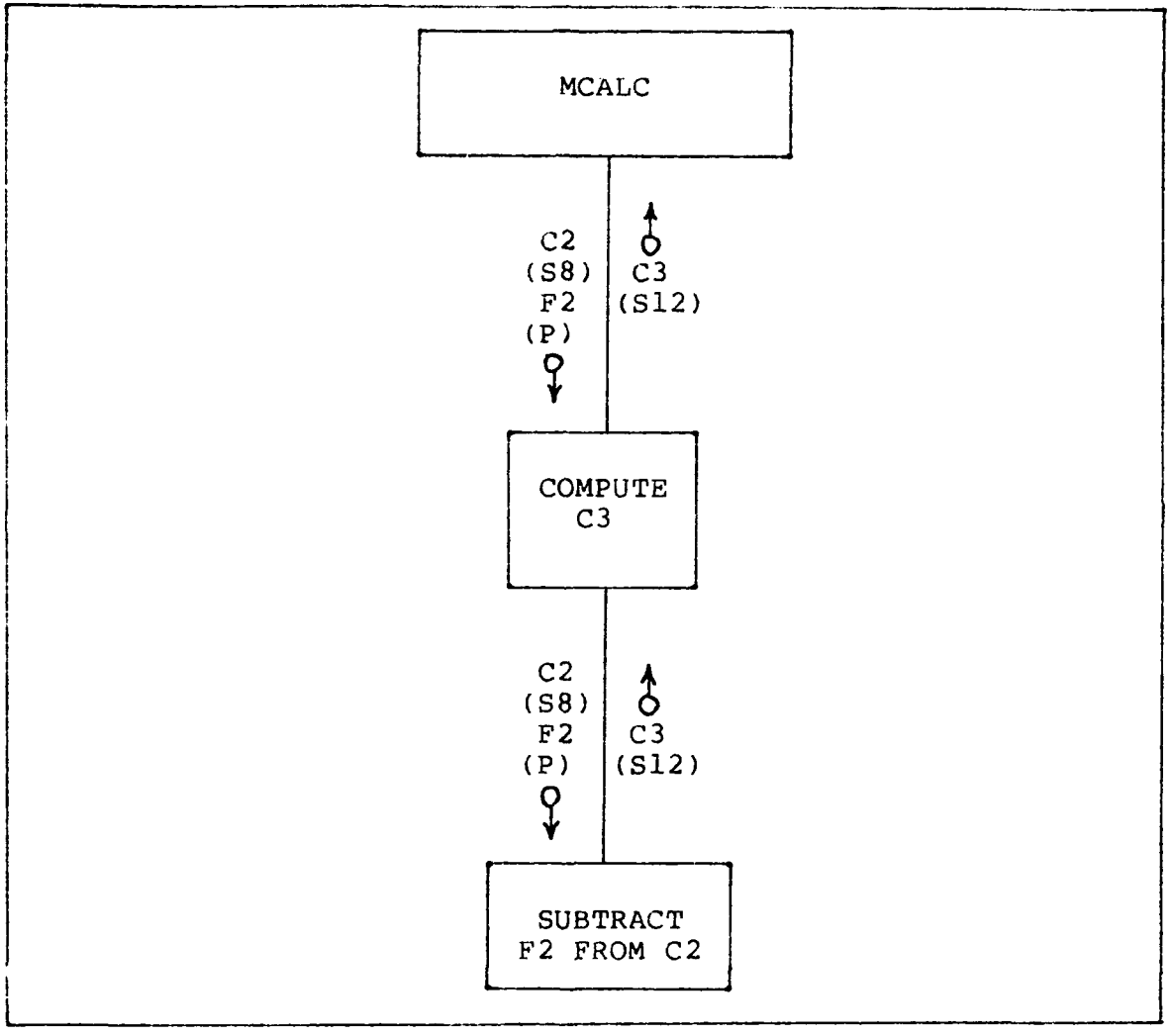


Figure 11. Cont.

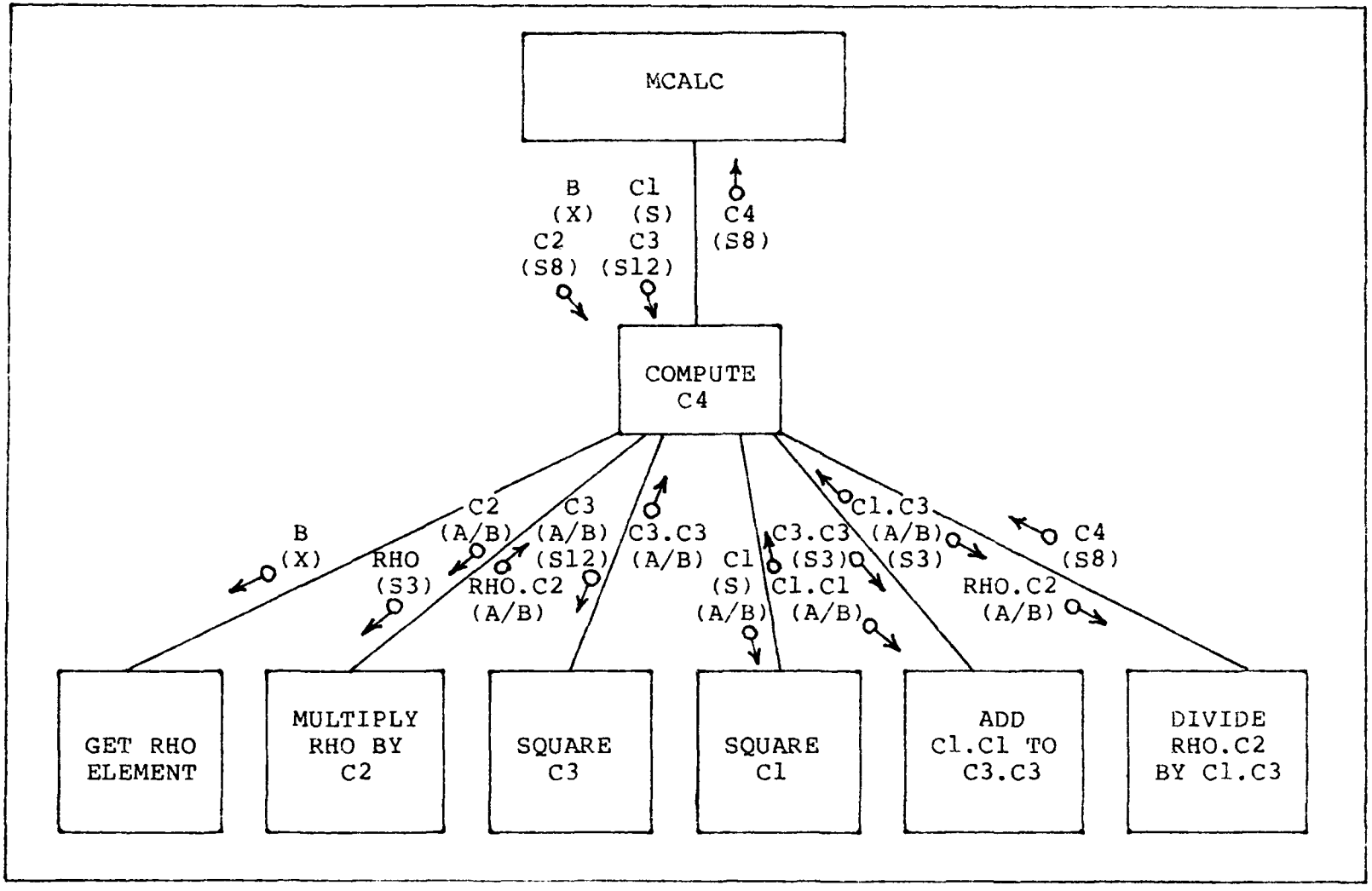


Figure 11. Cont.

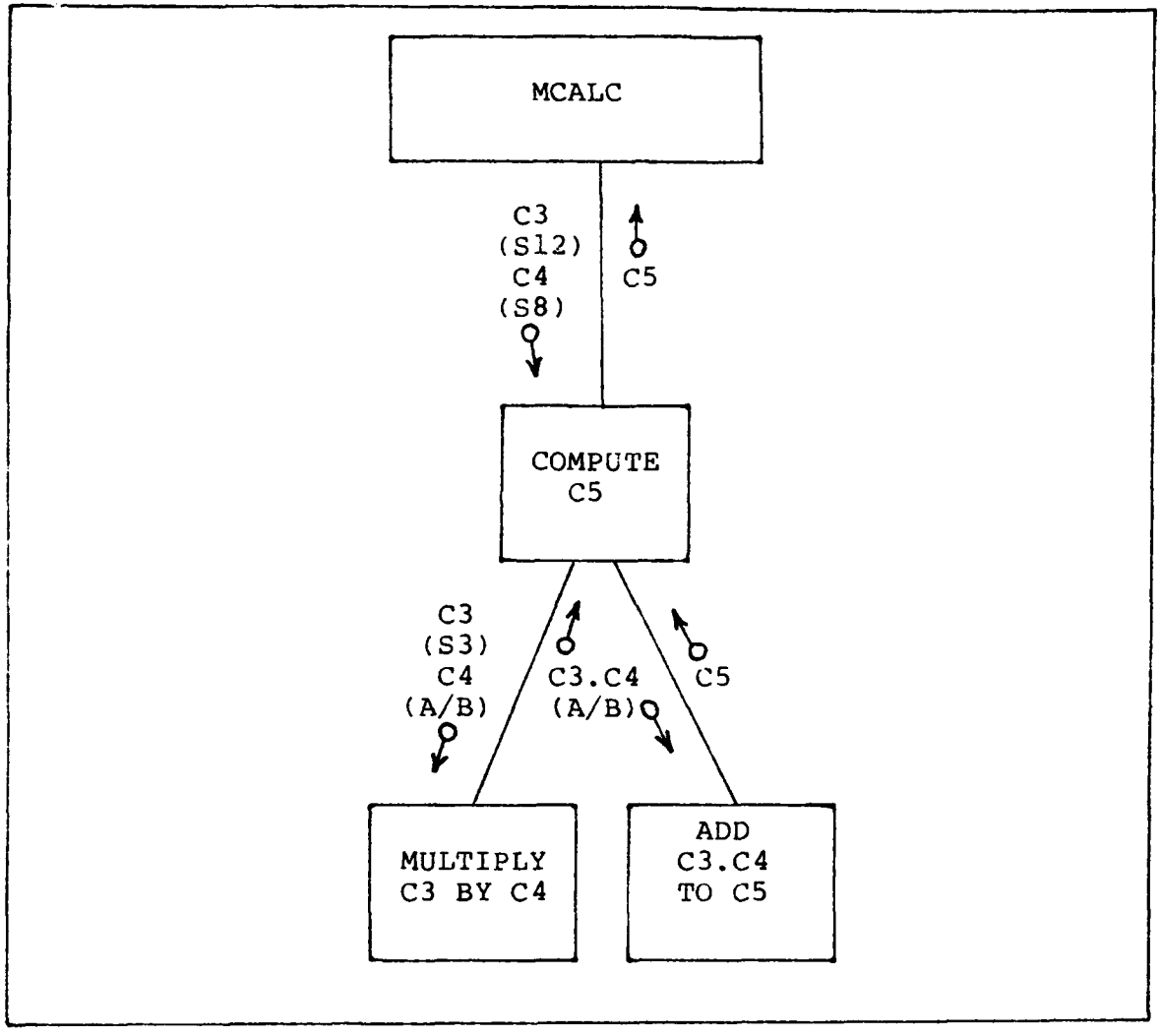


Figure 11. Cont.

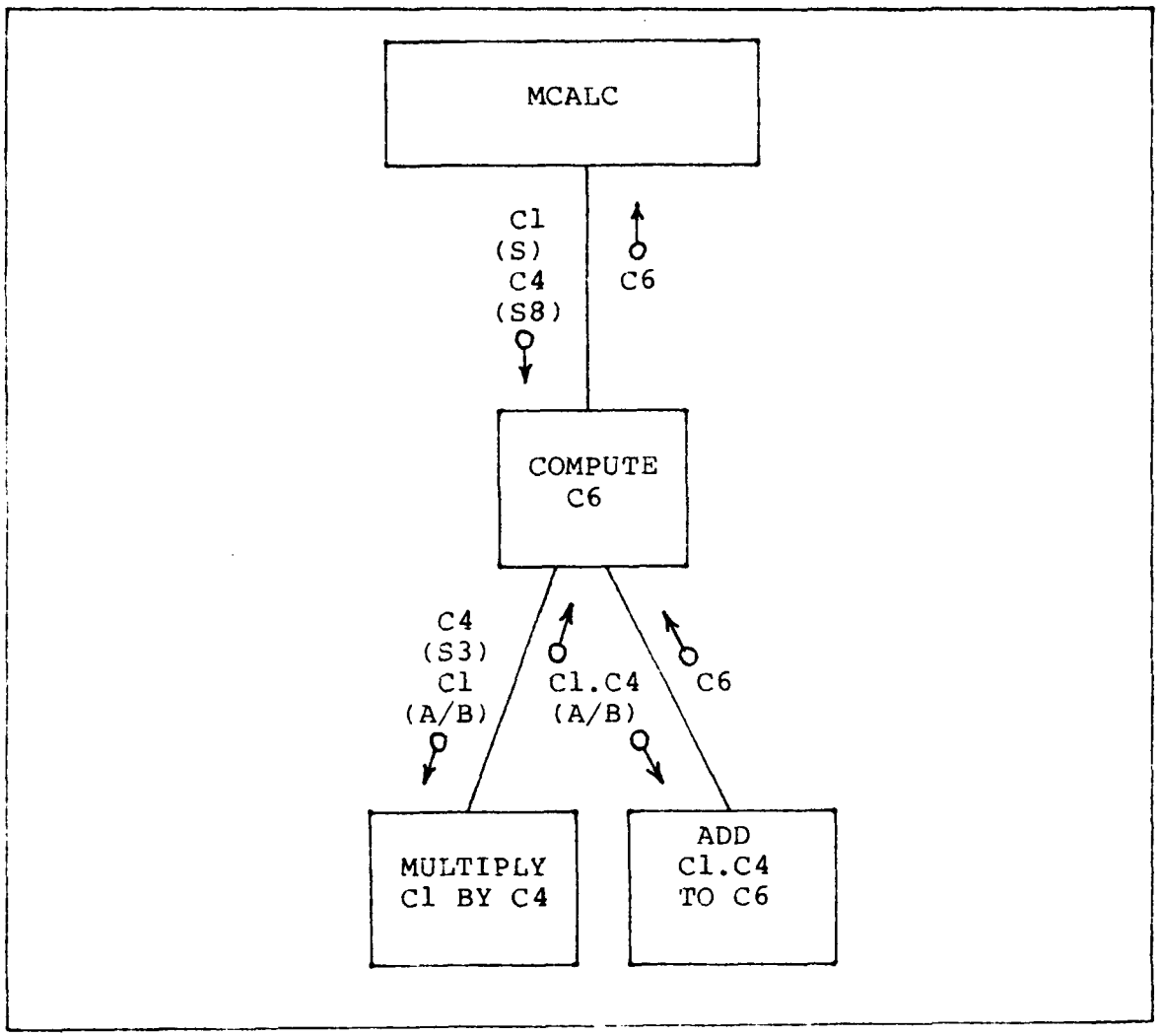


Figure 11. Cont.

a. Read C2 from TMP2 into A/B.
b. Put RHO element address into S3.
c. Call FMPY.
d. Save result (RHO.C2) in TMP2.
e. Read C3 from TMP3 into A/B.
f. Put C3 (TMP3) address into S3.
g. Call FMPY.
h. Save C3*C3 temporarily into C1C3 location.
i. Read C1 from TMP1 into A/B.
j. Put C1 (TMP1) address into S3.
k. Call FMPY.
l. Put C3*C3 (C1.C3) address into S3.
m. Call FADD (Floating Add).
n. Save C3*C3+C1*C1 in C1.C3.
o. Read RHO.C2 from TMP2 into A/B.
p. Call FDV (Floating Divide).
q. Save resulting C4 in TMP2.

6. Calculate C5. C5=C5+C3*C4
   a. Put C3 (TMP3) address into S3.
   b. Read C4 from TMP2 into A/B.
   c. Call FMPY.
   d. Read C5 address into S3.
   e. Call FADD.
   f. Save new C5 back in C5 location.
7. Calculate C6. C6=C6+C1*C4
   a. Read C1 from TMP1 into A/B.
   b. Put C4 (TMP2) address into S3.
   c. Call FMPY.
   d. Read C6 address into S3.
   e. Call FADD.
   f. Save new C6 back in C6 location.
8. Check for completion.
   a. Decrement loop counter (Y-reg).
   b. If the counter does not equal 0, go to step 2.
   c. Return to calling assembly language routine.

Implementation of MCALC

MCALC was implemented in microcode, using a short assembly language routine called ACALC to handle the interface between the FORTRAN-coded CALC and the microcoded MCALC. Listings for these three programs are in Appendix J.

The implementation of MCALC in microcode was very difficult because of the number of floating point operations required -- three adds, one subtract, seven multi-

plies, and one divide. This requirement created two major problems -- a subroutine return address problem and a WCS space problem.

As discussed in the previous chapter, the microcoded floating point routines in ROM cannot be used directly because of the problem of leveled subroutine calls in the M-Series machine. Only one return address can be stored in the SAVE register. This problem was solved in the LOADS microprogram by duplicating the subroutines in WCS and modifying them to return to a fixed address in the calling program. This direct return technique was possible in LOADS because only one call was needed to the floating point multiply routine and one call to the floating point add routine.

This direct return technique would also have worked in MCALC for the one divide, but not for the adds, multiplies, and the one subtract. The subtract routine is actually part of the the add, so it is also effectively called several times. If a subroutine is called more than once from more than one address in the program, then the return address must somehow be saved, or the subroutine must have some way of modifying a fixed return address. Another register could be used to store the extra return address, to augment the SAVE register, but there are no microinstructions to make the transfer into the SAVE register. The problem was finally solved by storing all the return addresses of the microprogram in a table and coding a

"jump" to the beginning of the table modified by an offset saved into the instruction register. Thus, in a sense the return address was stored in the instruction register.

The WCS space problem was created because of the apparent necessity to duplicate all of the floating point routines in WCS. Duplicating all four routines required 146 words of the 256-word WCS. This did not allow enough room for the rest of the program. This problem was solved by duplicating all but the divide routine, saving 53 words. This was enough to allow the 256-word microprogram to fit in WCS. The one divide was accomplished by microcoding instructions to load the divide arguments into their proper registers and main memory locations, and then returning to a macroinstruction to perform the divide. The divide macroinstruction was then followed by another macroinstruction which reinvoked the microprogram at the continuation point. This "trick" allowed control of the loop to remain at the microcode level, even though the divide was initiated by a macroinstruction.

Testing

Testing of MCALC consisted of a module test only. This test was run on the AFIT HP system using the special driver program CDRVR to provide the necessary inputs to MCALC (via CALC and ACALC). The test had two major purposes : (1) to verify correct output, and (2) to measure speed increase as a result of microprogramming.

Output Verification. The test data used for the output verification phase of the test was obtained from AFWAL/MLPJ personnel. Typical values for each of the equation parameters were chosen. The method of verification used was to simply compare the outputs of the microprogrammed version of the program to the non-microprogrammed version.

As expected, the program did not produce the correct outputs the first time, and debugging was necessary. Debugging was severely hampered by the size of MCALC. The Micro Debug Editor (MDE), which was very useful in the debugging of the LOADS microprogram, was much less useful here. If breakpoints are used in the debugging process, MDE requires almost half of the 256-word WCS space to operate. This meant that MCALC had to be segmented into overlays, and loaded into WCS in parts. This overlay technique of debugging was found to be very frustrating and prone to human error. Breakpoints in each overlay had to be carefully planned, so that the next overlay could be loaded. Segmenting the program into overlays also required keeping two separate versions. This led to several false indications when the two apparently identical versions (except for overlaying) gave different results. The debugging difficulty was compounded even further by the fact that MCALC was a loop.

Because of the problems of using the overlays with the MDE, the overlay debugging technique was largely abandoned

for a higher-level approach. Under this approach, the entire MCALC microprogram was loaded, and the MDE was not used for setting breakpoints. Inputs were modified in the FORTRAN driver to detect corresponding changes in the microprogram output. If it was necessary to examine a micro-level register, the microcode was modified slightly to pass the register value back to the FORTRAN driver through a main memory address. The MDE was still useful for making small changes to microcode. This saved editing and reassembly of the microprogram source, and also kept the source and object files free of debugging code. This higher-level debugging approach was successful, and the microprogrammmed version finally produced the expected outputs, completing the first phase of the module test.

Speed Measurement. The speed measurements of MCALC and the FORTRAN loop replaced by MCALC were accomplished using executive calls to read the system clock. This was the same technique that was used on the LOADS microprogram. The routines were called 100 times from a loop to get accurate timing measurements. The measurements showed the microprogrammed routine to be about ten per cent faster than the FORTRAN routine.

As in the speed improvement of the LOADS microprogram, this ten per cent speed increase was significantly less than the gains of six to ten times (Ref. 5:98) or two to twenty times (Ref. 9:49) reported in the literature. Close analysis of the assembly language generated for the FORTRAN

routine provided the answer to this apparent disparity. Totaling the instruction times for floating point and non-floating point instructions showed that 66 per cent of the routine's execution time was spent in the floating point instructions. Since the microcoded version of the program used these same floating point routines, no speed improvement could be made to 66 per cent of the program. This meant that even if all the non-floating point instructions could have been eliminated, the speed gain would still have been only 34 per cent! Thus, the gain of ten per cent was reasonable for this particular program.

Summary

This chapter covered the requirements, design, implementation, and testing of a microprogram called MCALC, designed for use in the laser materials modeling program. The application of the microprogram showed a ten per cent speed improvement in the refractive index calculation routine of the modeling program. This small improvement was due to the high ratio of floating point to non-floating point instructions in the program. This improvement was not great enough to show an operational improvement of the program, and thus was not tested on the operational machine.

VI. Automating the Tuning Process

Introduction

The microprogramming tuning technique used in the wind tunnel control program and the laser materials modeling program is largely a manual technique. Except for the generation of the program activity profile, all of the tuning processes must be accomplished manually by the programmer. This technique, while effective, is slow, costly, and requires microprogramming expertise. These disadvantages motivate the study of automatic tuning techniques. The purpose of this chapter is to review the research that has been done in the area of automatic tuning, and with this background, discuss the feasibility of developing an automated tuning system on the AFIT HP 21MX computer.

Background

The literature search done for this thesis investigation revealed that several researchers (Refs. 13-19) had done work in the area of automatic tuning of computer architectures. Three different tuning approaches from the literature are presented here.

Tuning Approach #1. The first approach presented is one by K.A. El-Ayat and J.A. Howard (Ref. 17). In their approach the tuning process is divided into three major steps: (1) performance monitoring and measurement, (2)

analysis of the data of the first step, and (3) synthesis of microprograms to improve deficiencies found in the second step. The goals of this approach are "significant improvement in performance, low implementation cost (overhead) and minimal human intervention" (Ref. 17:86).

The performance monitoring and measurement step is essentially an enhanced version of the activity profile generation used in this thesis study. As discussed in Chapter III, the monitoring can be done with hardware, software, or microcode. Here, the step is done with a very short (eight lines) microprogram, presumably added to the instruction fetch routine. The result of the performance monitoring and measurement is an instruction trace and a trace of data referencing patterns. The instruction trace indicates where a program should be tuned, and the data trace indicates which data items should be stored in micro-level registers to eliminate main memory fetches.

The purpose of the analysis step is to analyze the data from the first step to determine where the program should be tuned. Two types of program segments are selected for tuning, loops and non-loops. The loop segment is a set of instructions which is terminated by a branch back to the first instruction. The non-loop segment can be terminated by either a branching or non-branching instruction. In the non-branching case, termination is indicated when the profile activity of that instruction is less than that of the preceeding instruction. If the segment is

terminated by a branching instruction, the branch cannot be back to the first instruction.

In the analysis algorithm, the execution profile is searched, and each instruction execution count is compared to a minimum preset threshold count. If the threshold is met, the instruction corresponding to the execution count is selected as the first instruction of the segment. Subsequent instructions are examined to determine the end of the segment and the segment type. A segment must also meet a preset minimum size threshold (minimum number of machine instructions). The resulting output of the analysis step is a set of program segments ordered by segment type, size, and execution frequency. This ordering assures that segments having the greatest potential for performance improvement are tuned first, since the WCS space may prevent the tuning of all the selected segments.

The final step of this tuning approach is the synthesis of the microprograms and the machine language instructions which invoke the microprograms. Program segments are taken in order from the analysis step, and checked to see if the corresponding microcode will fit in the WCS. If so, the first machine instruction of the segment is replaced with an instruction which invokes the microprogram. This instruction is followed by the segment operand addresses. Each instruction is translated into microcode using the instruction opcode as the translation key. The microcode is then loaded into WCS, ready for execution. Loop seg-

ments require extra microcode to initialize working micro-level registers, which store loop variables, frequently used operands, and intermediate results.

The synthesis step also includes microcode optimization. The optimization applied here eliminates unnecessary instruction and data fetches, makes use of local store and emit fields within microinstructions, eliminates redundant and negated microoperations, and uses parallel microoperations when possible.

Results of tuning experiments using this approach show that the speed of loop segments increase 4 to 8 times, and non-loop segments by 1.7 to 4 times. The speeds of the overall programs show a 30 to 45 per cent improvement.

Tuning Approach #2. The second tuning approach presented here is one by Philip S. Liu and Frederic J. Mowle (Ref. 18). It is actually four separate methods of tuning that they have studied: (1) "Static Loading of Inner Loops," (2) "Selective (and Static) Loading of Inner Loops," (3) "Dynamic Overlaying of Inner Loops," (4) and "User Aided." The first three methods consider only inner loops of programs as candidate segments for microprogramming. The candidate segments of the fourth method can be either loop or non-loop segments.

The first method requires the compiler to identify all the inner loops of the program. The loops are then converted to microcode in the order that they appear in the program. Data items within the loops, both variables and

constants, are mapped into available micro-level registers. If not enough registers are available, the most frequently used data items are mapped first, and the remaining items are accessed from main memory. The conversion process, which can be done at the source or object code levels, continues until the WCS is filled. The major drawback of this first method is that the WCS can be filled before all the loops have been converted. Since the loops have been taken in the order that they appear in the program, some time-consuming loops may be omitted.

The second method remedies the drawback of the first method by requiring the compiler to assign priorities to the inner loops. Loops are then converted and loaded into the WCS on a priority basis. The priority of an inner loop is equal to its number of outer loops. The assumption here is that the inner loop with the greatest number of outer loops will be executed the most times, and should be given the highest priority. Inner loops with equal priority are converted to microcode in order of size, the one with the most object instructions taken first. With this second method all tne inner loops may still not fit in the WCS, but at least the most important ones are loaded first.

The third method insures that all inner loops of the program can be loaded into the WCS, but not all at the same time. This method works like a cache memory system where the main memory is divided into blocks, and a block is loaded into the faster cache memory when it is needed. In

this third method, all the inner loops are converted to microcode, given an identification number, and stored in main memory. When a loop is needed, the identification number of the one currently in the WCS is checked. If the needed loop is not in the WCS, then it is loaded over the one currently in the WCS. The major problem with this method is the overhead of swapping microcode in and out of the WCS. This overhead can be quite high because the WCS word length is usually greater than that of the main memory word, requiring two, three, or even more main memory word tranfers for one WCS word. The speed gain of the microcoded loops has to be great enough to offset this overhead.

The first three methods assume no a priori knowledge about the execution of the program. The fourth method assumes that the user has such knowledge about the program. This method allows the user to specify the program segments to be microcoded and the order in which they are microcoded. All the microcoded segments can be initially loaded into the WCS as in the first and second methods, or they can be dynamically overlayed as in the third method.

All four of the above methods were tested with six arbitrarily-chosen FORTRAN programs. The resulting speed gains are shown in Figure 12 (Ref. 18:Fig. 6) as functions of the WCS size. As shown, the fourth method produced the best program improvement, because of the human intervention. The second method, however, did almost as well with no human intervention. With a small WCS size, the third or

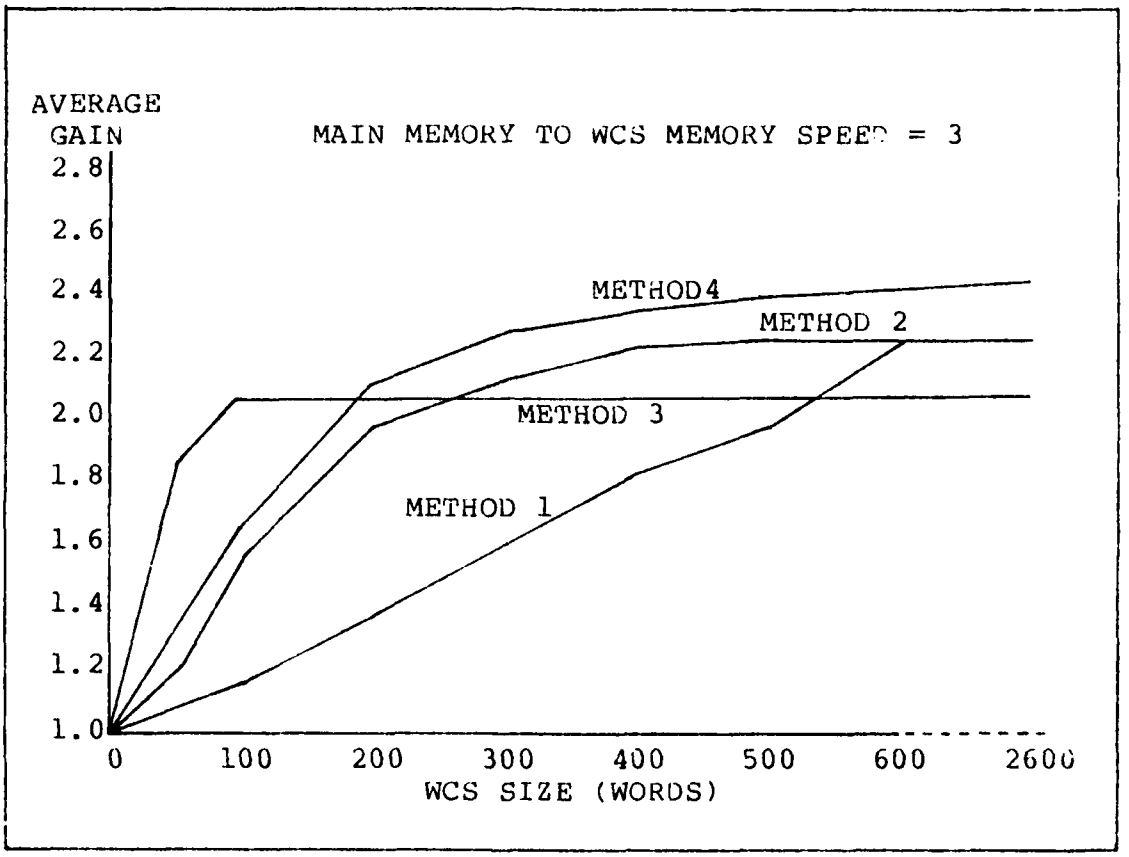
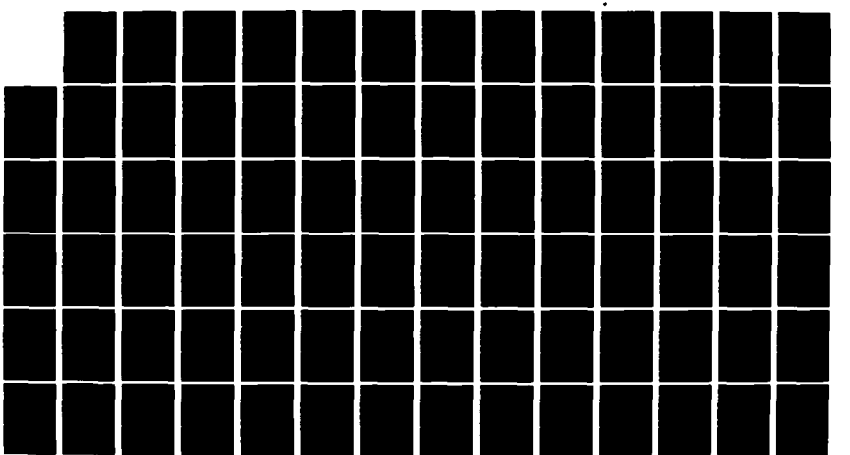


Figure 12. Average Gain of Test Programs

AD-A124 853 APPLICATIONS DIRECTED MICROPROGRAMMING ON A MINICOMPUTER SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 2/2

UNCLASSIFIED G A SCHOON DEC 82 AFIT/GCS/EE/82D-31 F/G 9/2 NL

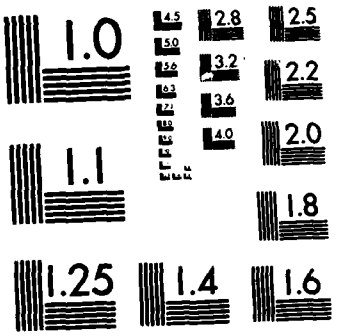


END

FILMED

TIC

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

overlay method did the best. Liu and Mowle recommend a combination of the third and fourth methods, the user aided and dynamic overlay methods.

Tuning Approach #3. The third tuning approach is one used in a system designed and implemented by K. Sakamura, T. Morokuma, H. Aiso, and H. Iizuka (Ref. 19). A model of the system is shown in Figure 13.

The model shows that the system consists of a computer, a monitor, an analyzer, a data base, and a feedback mechanism. The computer executes the program (or problem). The monitor collects information on the relative frequencies of machine instructions, sequences of instructions (serial dependencies), and address and data values. The analyzer uses this information obtained by the monitor to determine which segments of the program should be microcoded in order to speed up the program execution. The analyzer then synthesizes these new microcoded instructions. The feedback path is used to write the newly synthesized microprograms into the WCS, thus tuning the architecture of the computer. The data base for learning stores information about previous iterations of the tuning process. The analyzer can refer to this information in order to minimize the number of iterations.

An experimental system has been implemented using an HP 2100 computer with a 1K X 24-bit control store (0.25K ROM and 0.75K WCS). The monitor is a DYNAPROBE 7900+8000 hardware monitor, and a PDP-11V03 is used as the analyzer

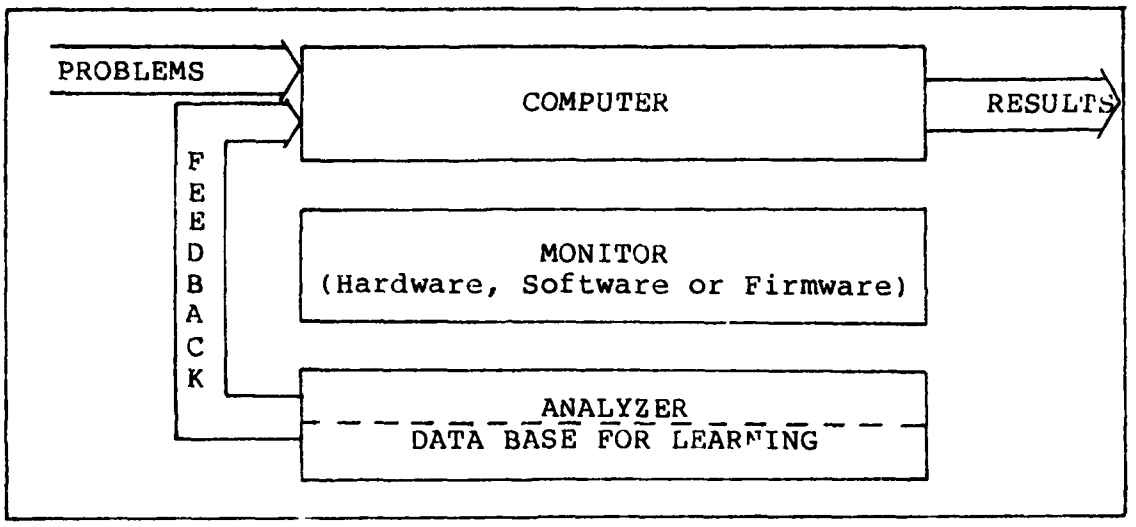


Figure 13. Model of an Automatic Tuning Mechanism

and synthesizer. A block diagram is shown in Figure 14.

As the program under test executes, the DYNAPROBE monitors the execution and feeds the information directly to the PDP-11V03. The PDP-11V03 analyzes the execution information, synthesizes the new instructions, and passes these new instructions to the HP 2100 through the I/O interface.

Since separate hardware is used for both the monitor and analysis functions, there is very little, if any, overhead in the tuning process. The result of the experimental system is a 30 to 60 per cent improvement in execution time of the tuned programs over the originals.

Review of the Three Approaches. The three approaches discussed are quite different from each other, but they share two common steps: (1) automatic determination of the program segments to microprogram, and (2) automatic synthesis of the microprograms.

The first approach uses a microprogram to precisely monitor program execution. The program is divided into loop and non-loop segments, and the execution data is used to determine which segments to microprogram. In the second approach the process of determining which segments to microprogram is simplfied by choosing only inner loop segments or other segments specified by the user. The third approach uses a hardware monitor to obtain program execution data, and uses a separate computer to perform the analysis and determine which segments to microprogram.

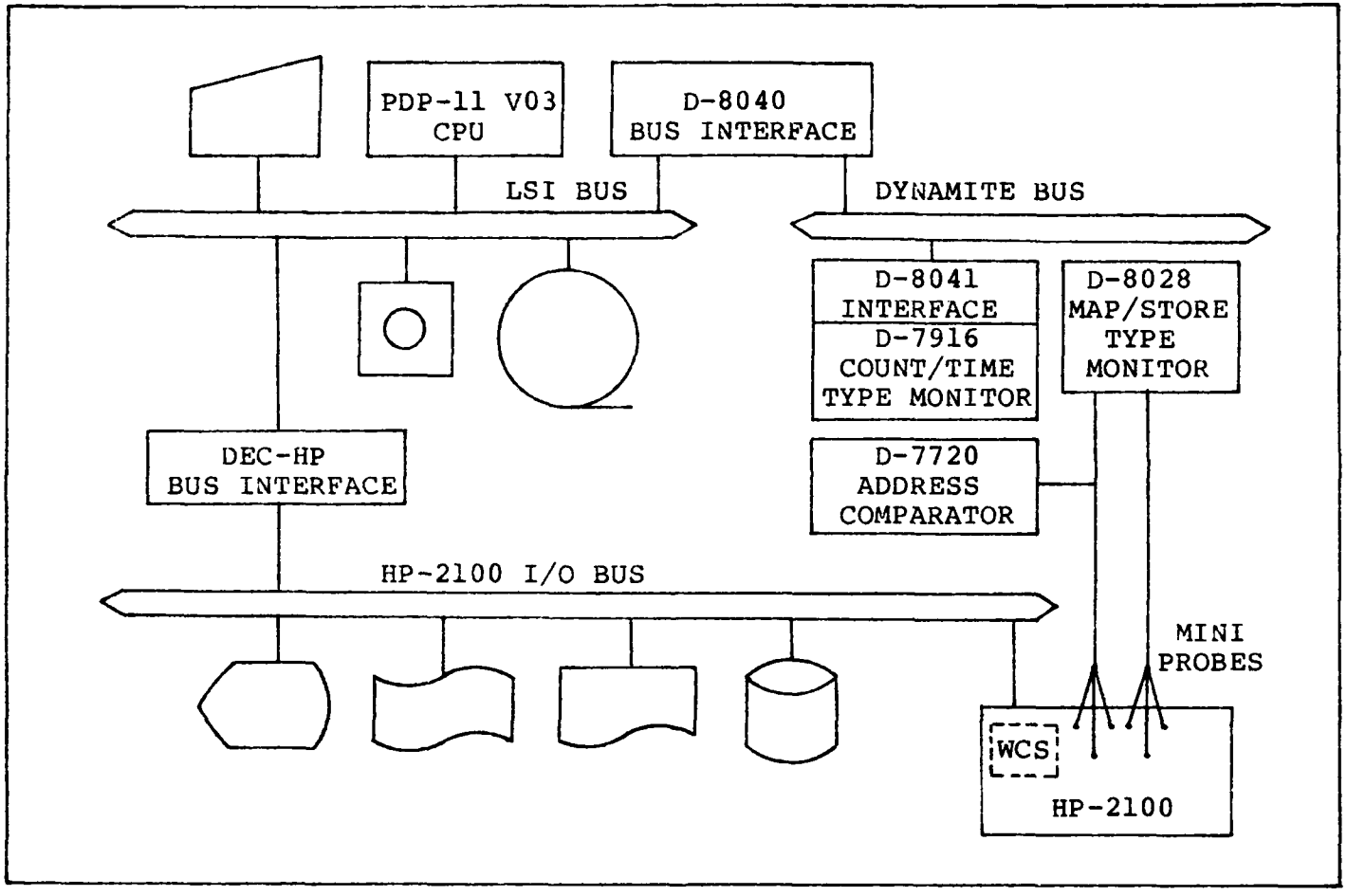


Figure 14. Functional Configuration of the Experimental System

This approach is the most sophisticated of the three, but also requires the most hardware. The first approach is next in sophistication, but requires modification of the computer's microcoded fetch routine. The second approach is by far the simplest and requires no extra hardware or special modifications to the computer firmware.

While all three approaches use automatic microprogram synthesis, only the paper on the first approach covers the actual algorithm used to perform this step. The software which implements the algorithm resides in the machine running the program, and the synthesis step is performed in an "off-line" mode. The third approach uses a separate computer to perform this step, the same machine that performed the analysis. In this approach the synthesis (and analysis) is performed while the application program is running, and the new microprograms are transferred back to the application machine through a feedback loop. Thus, the synthesis is an iterative process performed in an "on-line" mode. Details of the process in the second approach are not given. Again, the third approach seems to be the most sophisticated at the expense of more hardware.

The performance results of the three approaches are similar. The first approach showed performance improvements of 30 to 45 per cent, and the third approach showed improvements of 30 to 60 per cent. The second approach had similar gains, although they are given as ratios of non-tuned to tuned execution times, rather than percentages.

Automating the AFIT HP 21MX System

The background information on the three automatic tuning systems provides a good perspective for examining the feasibility of an automated tuning system on the AFIT HP 21MX system. The following discussion is intended to present the general requirements and some possible approaches to developing such a system.

General Requirements. The general requirements can be given in terms of user-system interface, system input and output, and performance objectives.

The users of this system are expected to be competant programmers in higher level languages, mainly FORTRAN, since this is the major higher level language used on the HP 21MX at Wright-Patterson. They may or may not have experience with HP 21MX assembly language, and probably do not have microprogramming experience. The tuning system should be designed with these experience levels in mind. The system does not have to be totally automatic with no user interaction, such as the one in the third approach discussed. In fact, an interactive system may be preferable, as suggested by "user aided" method in the second approach. The system should, however, be user-friendly and should make the details of the microprogramming and the micro-level architecture as transparent as possible to the user.

The inputs to the system consist of the all of the

available files associated with the program (FORTRAN or assembly language) being tested, plus interactive inputs from the user. The program files include the following: source, relocatable object, memory image, listing, and load map files. All of these files may not be needed to accomplish the tuning process, but are listed anyway as possible inputs. The one output of the system is an executable memory image file of the tuned program.

A performance objective is an important requirement for the system, but it is difficult to specify a program speed improvement figure that the system should be able to meet. The amount of improvement of a given program is largely dependent on the characteristics of that program. This is true for manual tuning as well as automatic tuning. A 25 to 30 per cent improvement is probably a reasonable performance objective for an automatic system as indicated by the results of the three approaches discussed. Anything below this is probably operationally insignificant for most programs.

Possible Approaches. As discussed, the three approaches share two common steps in the tuning process: (1) automatically (with possible user-interaction) determining which segments of the program to microprogram, and (2) automatically synthesizing the microprograms. Possible approaches to automating the AFIT system are discussed in terms of accomplishing these two basic tuning steps.

The activity profile generator program (ACTV) par-

tially satisfies the requirement to determine which program segments to microprogram. ACTV in its present form divides a program under test into 50 equal intervals and determines a profile count for each interval. These equally divided intervals do not, however, correspond to the logical segments of the program, the loops and non-loops, for example. The profile counts for these equal intervals must be converted into counts for the logical segments. This requires first the determination of the address boundaries of the logical segments, and then the correlation of these boundaries to the profile interval boundaries.

An example may help explain the process. Figure 15 contains a block diagram and partial execution profile of a program with three segments -- a non-loop followed by a loop, which is followed by another non-loop. The address boundaries in octal for the three segments are 40000 to 40250, 40250 to 40330, and 40330 to 40620 respectively. Finding the segment boundaries requires an algorithm which analyzes the branching instructions of the program. The example shown contains at least one branching instruction, a "jump" from the end of the loop segment back to the beginning. No software currently exists at AFIT to perform the segment-bounding task on the HP system, but the software should not be difficult using an existing algorithm. El-Ayat and Howard, authors of the first approach discussed, describe one algorithm for finding segment boundaries (Ref. 17:86). Their algorithm requires a very pre-

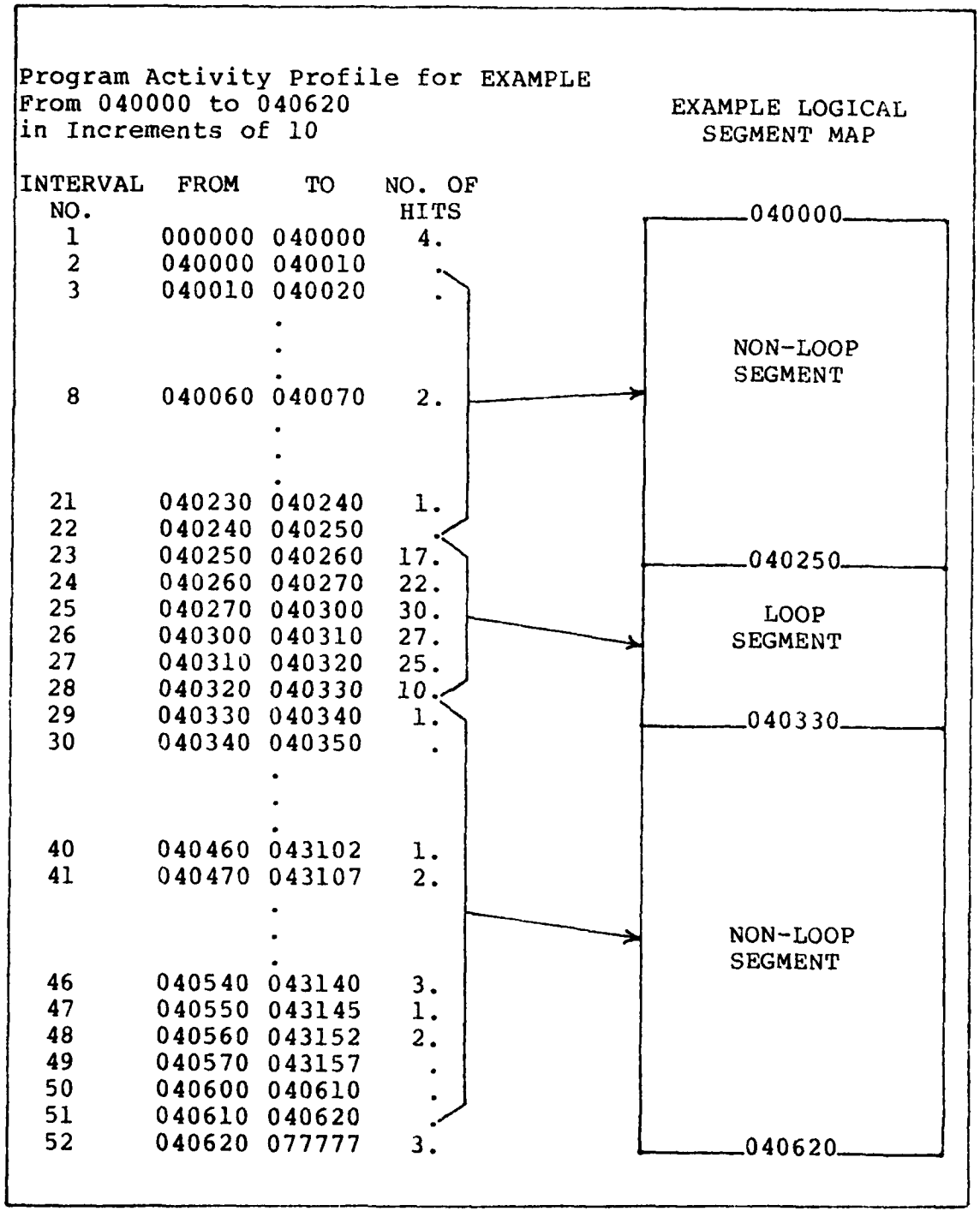


Figure 15. Mapping Profile Interval Counts to Logical Program Segments

cise program activity profile, however, and would have to be modified to work with the statistical profile provided by ACTV. Liu and Mowle, the authors of the second approach, mention another technique called "control flow analysis" for detecting loops in a program, but do not give details of the technique (Ref. 18:817). User interaction may also be beneficial in finding segment boundaries of a program.

When the segment boundaries are found, the conversion of interval counts to segment counts can be done. The activity profile in the figure shows counts or "hits" for several of the 52 intervals (50 intervals in the program range and two outside). The bracketed "hits" show the mapping of the profile intervals to the logigal segments of the program. From a mapping such as this, the profile counts for each logical segment can be determined, and a decision on which segments to microprogram can be made. This process can easily be done by software given all the input information shown in the figure.

The reader should note that this is a contrived example with the interval and segment boundaries chosen to allow a perfect mapping. In practice, this does not usually happen. A profile interval can overlap a segment boundary, making it difficult to determine which segment receives the profile count. This is probably not major problem, because other segment counts can be used to determine the probable correct segment. If the program is very large, the inter-

vals of the profile can actually be larger than the logical segments, since the profile generator program presently allows only 50 intervals. Several logical segments could then occur in one interval, again making a correct mapping of the count difficult. Two possible solutions to this problem exist. The number of profile intervals can be dynamically adjusted (within main memory limits) to match the program size, or several profiles can be run with each "looking" at a different portion of the program. This latter approach was used in the manual tuning of the wind tunnel and laser materials programs.

Another approach to determining which program segments to microprogram is to choose only loop or inner-loop segments as in the second approach of the background information. This approach eliminates the need for any type of activity profile generation and the problem of mapping profile intervals to program segments. The segments must still be identified, but this has to done anyway. This approach has the advantage of simplicity, and the results from the three approaches discussed shows that it compares favorably with the others. Also, the analysis of the two programs in this thesis study supports the theory that loops account for much of the program activity, and are the best candidates for microprogramming.

From a user point of view, determining which segments of a program to microprogram is the easier of the two basic steps in the tuning process. The concept of program seg-

ments, such as loops and non-loops, and their execution times is nothing really new to the higher-level language programmer. The second step of synthesizing the microprograms is a much more unfamiliar task, and requires familiarity with the architecture of the machine and assembly language and microprogramming expertise. Thus, the automation of this step is even more important. An example is used here to explain the synthesizing process.

Figure 16 shows the synthesis process for two macroinstructions from the manually-tuned wind tunnel subroutine SPEED. The two macroinstructions, "LDA .YZT" and "STA ..YZT", are shown along with the two "BSS" pseudo-instructions, which define memory locations for .YZT AND ..YZT. The function of these instructions is to simply load the "A" register with the contents of .YZT and store that contents into ..YZT (i.e., ..YZT = .YZT).

The figure shows the breakdown of the instructions' machine code into four fields -- D/I, opcode, Z/C, and argument relative address (all numeric values in octal). The opcode and argument relative address are self-explanatory. The D/I is a bit indicating whether the argument relative address is used directly or indirectly. The Z/C bit indicates whether the argument address is relative to page zero of memory or the current page. For these two instructions, both addresses are direct and relative to the current page -- page 42000 as indicated by the instruction addresses.

The breakdown of the machine code is the key to the

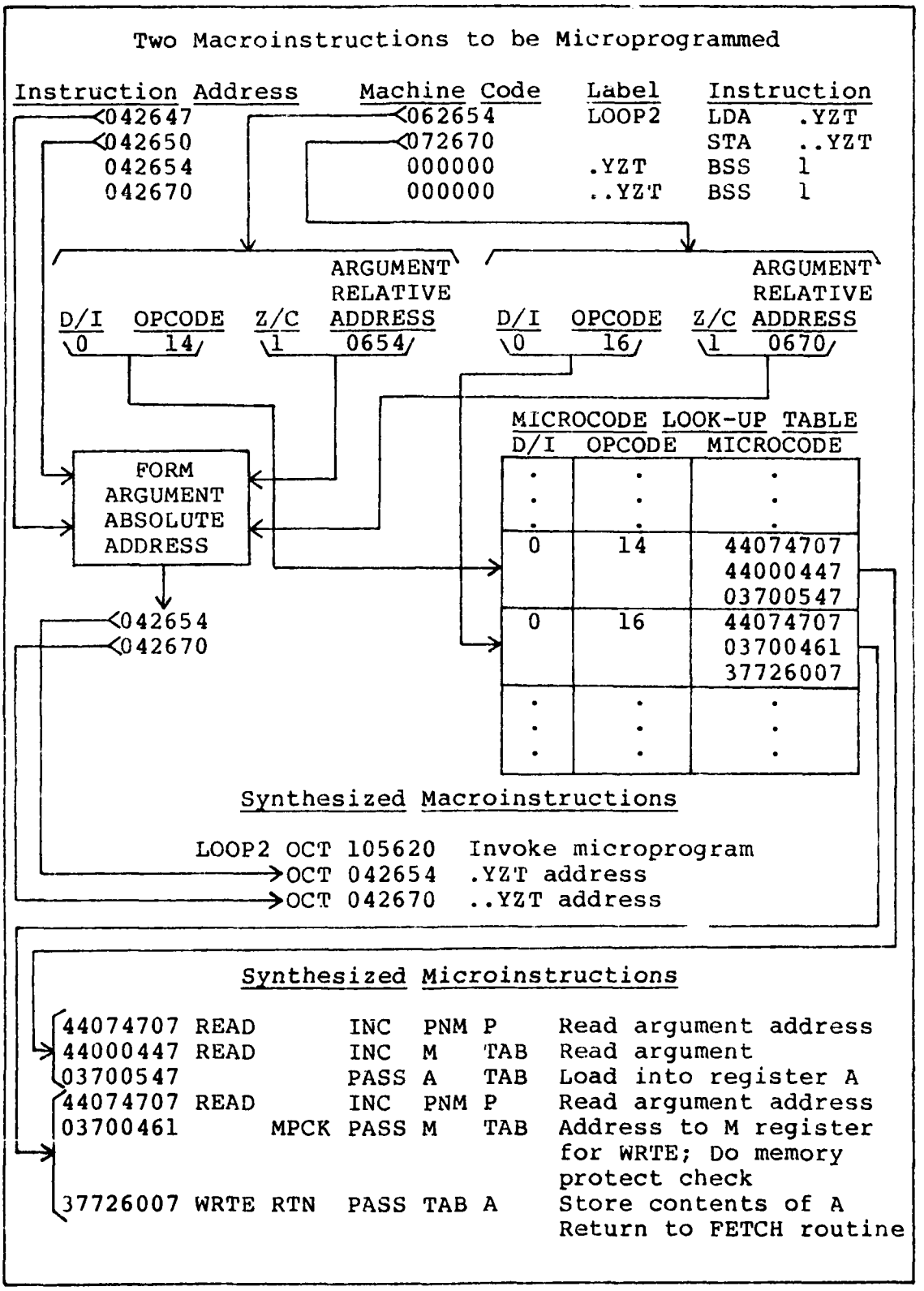


Figure 16. Microprogram Synthesis Example

synthesis process. The D/I and opcode fields can be used as an index into a microcode look-up table to obtain a sequence of microinstructions which will replace the original macroinstruction. This partial look-up table is derived indirectly from the microcode for the basic instruction set stored in control store ROM. A listing for the entire basic instruction set of the HP 21MX is available in Appendix E of Reference 32. Table XI shows the microinstructions for the memory reference group instructions, such as the LDA and STA instructions in the example. The microcode in Table XI cannot be used directly to translate a macroinstruction to a microroutine. The reason for this is that the macroinstruction is performed partially by the microcoded fetch routine (shown at the bottom of Table XI) and partially by the macroinstruction's microroutine. The LDA microroutine, for example, consists of only one microinstruction as shown in Table XI (LDA and LDB are shown as LD*). The fetch routine in this case performs the major part of the instruction. Another reason the microcode from the table cannot be used directly is that the microoperations within the microinstructions often perform operations which are conditional on information in the instruction register. When the macroinstructions are replaced by microcode, they are no longer fetched, and are not stored into the instruction register. The microroutines stored in the look-up table must compensate for the lack of the instruction fetch and the information in the instruction

TABLE XI
HP 21MX Memory Reference Group Microroutines and FETCH

Macroinstruction	Microroutine					Comments
AND,I	JSB				INDIRECT	RESOLVE INDIRECT ADDRESS
AND			PASS	L	A	L <- A
		RTN	AND	A	TAB	A <- T/A/B AND L
CP*,I	JSB				INDIRECT	RESOLVE INDIRECT ADDRESS
CP*			PASS	L	CAB	L <- A/B
			XOR		TAB	T-BUS <- T/A/B XOR L
	JMP	CNDX	TBZ		FETCH	JUMP TO FETCH IF EQUAL
		RTN	INC	P	P	P <- P+1 IF NOT EQUAL
XOR,I	JSB				INDIRECT	RESOLVE INDIRECT ADDRESS
XOR			PASS	L	A	L <- A
		RTN	XOR	A	TAB	A <- T/A/B XOR L
IOR,I	JSB				INDIRECT	RESOLVE INDIRECT ADDRESS
IOR			PASS	L	A	
		RTN	IOR	A	TAB	A <- T/A/B IOR L
ST*,I	JSB				INDIRECT	RESOLVE INDIRECT ADDRESS
ST*		MPCK	PASS		M	MEM PROTECT CHECK ADDRESS
	WRTE	RTN	PASS	TAB	CAB	T/A/B <- A/B; WRITE

TABLE XI (cont.)

```
AD*,I     JSB                      INDIRECT   RESOLVE INDIRECT ADDRESS
AD*                    PASS L      CAB        L <- A/B
          ENVE RTN     ADD  CAB    TAB        A/B <- T/A/B PLUS L

JSB,I     JSB  IOFF                INDIRECT   DISABLE INTERRUPTS
JSB            MPCK PASS          M          MEM PROTECT CHECKS ADDRSS
          WRTE      PASS TAB      P          T/A/B <- RTN ADDRSS;WRITE
               RTN  INC  P        M          P <- M+1

ISZ,I     JSB                      INDIRECT   RESOLVE INDIRECT ADDRESS
ISZ            MPCK PASS          M          MEM PROTECT CHECKS ADDRSS
                    INC  S1       TAB        S1 <- T/A/B + 1
          WRTE      PASS TAB      S1         T <- S1; WRITE
          JMP  CNDX TBZ  RJS      FETCH      ZERO? NO, DONE
               RTN  INC  P        P          YES, P <- P+1

LD*,I     JSB                      INDIRECT   RESOLVE INDIRECT ADDRESS
LD*            RTN  PASS CAB      TAB        A/B <- T/A/B


               Macroinstruction Fetch Routine
FETCH     READ FTCH INC  PNM      P            M<-P; P<-P+1; READ INSTR
               ION                             ENABLE INTERRUPTS
               CLFL PASS IR       TAB          IR<- T/A/B; CLR FLAG FF
          READ JTAB INC  CM       ADR          JUMP THRU TABLE; LOAD M
                                               IF MEM REF GROUP
```

register.

The microroutines for the LDA and STA instructions in the example are each three microinstructions long. The mnemonics for the microcode from the look-up table are shown at the bottom of Figure 16. The "RTN" microoperation of the last microinstruction is not actually encoded in the table. It is added to the last microinstruction to transfer control back to the macroinstruction level.

The Z/C and argument relative address fields are used to form an argument absolute address. If the address is relative to page zero, the relative and absolute addresses are equivalent. If the address is relative to the current page, the current page address is added to the relative address. The example shows the formation of the .YZT and ..YZT absolute addresses. These addresses are sequentially annexed to an argument list following a synthesized macroinstruction. This macroinstruction invokes the synthesized microroutine, and along with the argument list, replaces the original macroinstructions.

Synthesizing the microroutines in the manner described usually does not produce optimal microcode. Optimization should be performed as another step of the synthesis process, as in the first tuning approach discussed. Frequently used argument addresses should be removed from the argument list and stored into available micro-level registers. The address can then be accessed by a register transfer instead of a main memory read, saving at least one

microinstruction. Microinstructions which access main memory should be followed by microinstructions which do not access the memory data register, the "T" register. Memory reads and writes require two microcycles, and such instructions cause a "processor freeze" (Ref. 32:3-14) until the read or write is complete. At least three "freezes" occur in the synthesized microroutine of the example, but nothing can be done in this case. Parallel microoperations should be used as much as possible. An example of this is the addition of the "RTN" to the last microinstruction of the example, rather than making it a separate microinstruction. Redundant or negated microinstructions should also be eliminated (Ref. 17:87). These optimization checks can probably be done during the synthesis, but may be more easily done during a "second pass".

The final products of the synthesis step are the synthesized macroinstructions and microinstructions as shown at the bottom of Figure 16. The macroinstructions may directly overwrite the ones they are replacing in main memory or in the memory image file, assuming no relocation or operating system problems exist. The microinstructions may be written to the WCS or to a file using existing microprogram utilities.

The synthesis step, while more difficult than the analysis step, is staightforward and is quite adaptable to an automated tuning system. Probably the most difficult problem is the building of the microcode look-up table,

since this is largely a manual process. The documented microroutines for the HP 21MX base instruction set can be used as a basis for this table, but require substantial modification because of the elimination of the fetch instruction as discussed earlier. Although building the table presents a substantial manual microprogramming effort, it can be done and is not considered a major obstacle to automating the synthesis step.

Summary

This chapter has dealt with the feasibility of implementing an automated tuning system on the AFIT HP 21MX computer. Three automatic systems from the literature were presented to provide background on the tuning process and different approaches to the implementation of such a system. The general requirements for an AFIT system were discussed in terms of user-system interface, system input and output, and performance objectives. Finally, possible approaches to automating the system were discussed in terms of accomplishing the two basic steps of the tuning process -- determining which segments of the program to microprogram, and automatically synthesizing the microprograms. Although many of the implementation details have not been discussed here, an automated tuning system on the AFIT HP 21MX is certainly considered feasible.

VII. Results, Conclusions, and Recommendations

Introduction

This thesis study focused on the performance improvement of HP 21MX application programs using microprogramming tuning techniques. Routines from two application programs were chosen as candidates for microprogramming as a result of a survey of HP users at Wright-Patterson Air Force Base. The two routines chosen were a stress calculation routine for a wind tunnel control program and a refractive index calculation routine for a laser materials modeling program. Microprograms were written and applied at points in the routines indicated by activity profile analysis. The speed improvement of the resulting programs was then measured. The experience gained from tuning the two application programs and studies in the literature provided the background for investigating the feasibility of developing an automated tuning system on the AFIT HP 21MX. This chapter lists the results, conclusions, and recommendations of the thesis study.

Results

The following are considered the major results of the study:

1. The performance improvement of the wind tunnel stress calculation routine was about 31 per cent. This resulted in an operational improvement of 33 per cent in

the rod adjustment process of the wind tunnel control program by allowing the simultaneous adjustment of four rather than three rods. The routine was subsequently integrated into the operational version of the program. This may have been the first user-microprogram to be applied to an operational HP 21MX program at Wright-Patterson.

2. The performance improvement of the laser materials routine was about 10 percent, which was not enough to noticeably improve the waveform display for the user. This program showed the limitations of microprogramming in improving a routine with a large number of floating point operations. Writing and debugging the microprogram for this routine also provided experience working with large microprograms and leveled subroutines.

3. Both the wind tunnel and laser materials microprograms were developed using software engineering techniques. This resulted in structured microprograms that were documented much better than the original FORTRAN and assembly language candidate programs.

4. Work on the two application programs exposed at least two HP users at Wright-Patterson to user-microprogramming. These users will hopefully consider the possibility of applying microprogramming to future applications.

5. The investigation into the feasibility of developing an automated tuning system on the AFIT HP computer showed that such a system was feasible, and possible approaches to the development were discussed.

Conclusions

Based on the above results, the following conclusions are made:

1. Both of the application programs tuned involved floating point operations. Although two programs are hardly a large enough sample on which to base any hard conclusions, the inference here is that the HP 21MX application programs in greatest need of performance improvement are those with floating point operations. Ironically, these are the ones that can be helped the least with microprogramming on the HP 21MX-M Series or 21MX-E Series computers. The 21MX-F Series, however, has floating point operations implemented in hardware, and programs with a large number of floating point operations running on this series should benefit as well from microprogramming as programs with non-floating point operations.

2. Microcode can be structured and well documented using software engineering techniques. The complexity of a program, however, can increase significantly with the addition of microcode. The laser materials code segment, for example, was changed from a simple 10-statement FORTRAN "DO" loop to a "CALL" to a 38-statement assembly language routine, which invoked a 256-word microprogram! All this was done for a 10 per cent increase in speed! In this particular case the trade-off of speed versus complexity could not be justified, because the increase in complexity

greatly outweighed the overall benefit of the increase in speed. In the wind tunnel program, however, the routine was already at the assembly language level. Thus, much of the complexity already existed, and substituting the code to invoke the microprogram actually decreased the assembly language code. The microprogram was about one-half as long as the laser materials microprogram, and the speed increase was three times as much. The trade-off in this case was justifiable.

3. The manual tuning technique used in this thesis study is too cumbersome to become widely used at Wright-Patterson (or anywhere else). To use this technique a programmer must learn the HP assembly language, the micro-assembly language, their associated debugging tools, and the internal architecture of the system. The training time, along with application program analysis and micro-program development time represents a large investment with little guarantee of results.

Recommendations

The following recommendations are made as a result of this thesis investigation:

1. Since the application programs of this study both involved floating point operations, further study could focus on other types of applications where microprogramming might be of better benefit. Two possibilities are high-speed sorting and high-speed graphics. One ASD/ENAMA

program called MPASS (Refs. 26,36) could possibly use both of these capabilities. The program was not considered in this study because it had not been moved from the CDC computer to the HP 21MX. There are still no plans to move the program, but a high-speed sorting and graphics capability might seriously influence such a move.

2. Users with programs bound by floating point operations should consider upgrading to an HP 21MX-F Series machine. The hardware floating point operations of the F-Series machine are roughly 20 times as fast as the microprogrammed functions of the M-Series machine (Refs. 34:3-25 and 35:13-20). A letter received from the Hewlett-Packard Company indicates that no hardware floating point processors are available for the M-Series machine from either them or any other known source (Ref. 37).

3. Because of the drawbacks of the manual tuning technique used in this study, it is recommended that an automated system, as described in the previous chapter, be designed and implemented on the AFIT HP 21MX computer. In support of this effort, the upgrading to an F-Series should be seriously considered because of the floating point and microprogramming limitations of the M-Series machine. The initial work could, however, be done on the M-Series. An automated system would make tuning of application programs practical for all HP programmers. Without such a system, user-microprogramming has little future on this machine.

Bibliography

1. Rauscher, T.G. "Dynamic Problem-oriented Redefinition of Computer Architecture Via Microprogramming," IEEE Transactions on Computers, C-27 (11):1006-14 (November 1978).

2. Wilkes, M.V. "The Best Way To Design An Automatic Calculating Machine," in Manchester Univ. Comput. Inaugan. Conf.: 16, 1951.

3. Fagg, P. et al., "IBM System/360 Engineering," 1964 Fall Joint Computer Conference Proceedings: 205-31 (1964).

4. Snyder, David C. "Computer Performance Improvements by Measurement and Microprogramming," Hewlett-Packard Journal: 17-25 (February 1975).

5. Gordon, P. and S. Stallard, "Microprogrammed CPU Architecture Offers User-Alterable Minicomputer Performance," Computer Design: 91-100 (June 1978).

6. Steidle, John J. Microprogramming: A Tool to Improve Program Performance, AFIT Thesis AFIT/GE/EE/81D-56, 1981.

7. Microprogramming: A Way to Get Higher Performance from HP-1000 Computers. Product Application Note 281-1. Hewlett-Packard Corporation. Sep 1978.

8. Cook, R.W. and M.J. Flynn. "System Design of a Dynamic Microprocessor," IEEE Transactions on Computers, C-19: 213-22 (1970).

9. Frankenberg, R. "Unraveling the Mystery in User Microprogramming," Mini-Micro Systems, 46-49 (Jul 1977).

10. Hansen, G., "User-generated Microprograms Improve Mini Performance," EDN, 25: 145-9,151 (May 5, 1980).

11. Rauscher, Tomlinson G. and Ashok Kumar Agrawala. "Developing Application Oriented Computer Architectures on General Purpose Microprogrammable Machines," Proceedings of the 1976 National Computer Conference: 715-22 (1976).

12. Tucker, A.B. and M.J. Flynn. "Dynamic Microprogramming: Processor Organization and Programming," Comm. ACM 14, 240-50 (1971).

13. Abd-Alla, A.M. and David C. Karlgaard. "Heuristic

Synthesis of Microprogrammed Computer Architecture, IEEE Transactions on Computers, C-29, (8):802-7 (August 1974).

14. Abd-Alla, A.M. and Laird H. Moffett. "On-line Architecture Tuning Using Microcapture," 3rd Annual Symposium on Computer Architecture: 165-71 (January 19-21, 1976).

15. Luque, E. and A. Ripoll. "Tuning Architectures Via Microprogramming," Information Processing Letters, 11: 102-109 (October 20, 1980).

16. Luque, E., A. Ripoll and J.J. Ruz. "Dynamic Programming in Computer Architecture Redefinition," Euromicro Journal, 6:98-103 (March 1980).

17. El-Ayat, K.A. and J.A. Howard. "Algorithms for a Self-tuning Microprogrammed Computer," Micro 10 Proceedings: 85-91 (October 5-7 1977).

18. Liu, Philip S. and Frederic J. Mowle. "Techniques of Program Execution with a Writable Control Memory," IEEE Transactions on Computers, C-27 (9):816-827 (September 1978).

19. Sakamura, K., T. Morokuma, H. Aiso and H. Iizuka. "Automatic Tuning of Computer Architectures," AFIPS Conference Proceedings, 48: 499-512 (1979).

20. Meyers, Glenford J. Advances in Computer Architecture. New York: John Wiley and Sons, Inc., 1978.

21. Williams, Glenn. AFWAL/FIMN. Technical Discussions, April 1982.

22. Cain, Maurice R. "Mechanical Design and Control of a Variable Geometry, Adaptive Wall Test Section for a Pilot Transonic Wind Tunnel," unpublished interim technical report, AFFDL, Wright-Patterson AFB, OH. May 1979.

23. Phillippi, Conrad. AFWAL/MLPJ. Technical Discussions, June 1982.

24. Spitzer, W.G. and D. A. Kleinman. "Infrared Lattice Bands of Quartz," Physical Review, 121 (5):1324-35 (March 1, 1962).

25. Linder, Larry. AFFDL. Technical Discussions, May 1982.

26. Steidle, John. ASD/ENAMA. Technical Discussions, May 1982.

27. Meyers, Glenford J. Digital System Design With LSI Bit-slice Logic. New York: John Wiley and Sons, Inc., 1980.

28. Knuth, D.E. "An Empirical Study of FORTRAN Programs," Software -- Practice and Experience, Vol 1 (1974).

29. De Blasi, M., N. Fanelli, G. Giannelli, and G. Antoni. "Profile Finder, A Firmware Instrument for Program Measurements," Euromicro Newsletter, 3 (1):27-33 (January 77)

30. Weinberg, Victor. Structured Analysis. New York: Yourdon Press, 1980.

31. Spiegal, Murray R. Schaum's Outline of Theory and Problems of Advanced Mathematics for Engineers and Scientists. New York: McGraw-Hill Book Co., 1971.

32. Microprogramming 21MX Computers Operating and Reference Manual. Manual Part No. 02108-90008. Hewlett-Packard Company. August 1974.

33. HP 12978A Writable Control Store Manual. Manual Part No. 12978-90007. Hewlett-Packard Company. February 1976.

34. HP 21MX Computer Series Reference Manual. Manual Part No. 02108-90002. Hewlett-Packard Company. May 1974.

35. HP 1000 E-Series and F-Series Computer Microprogramming Reference Manual. Manual Part No. 02109-90004. Hewlett-Packard Company. July 1978.

36. Hill, Gary A. "An Introduction to the Multiple Penetrator and Site Simulator (MPASS) Program," ENADD Technical Note, ASD, Wright-Patterson AFB, OH. January 1981.

37. Apple, Charlene. Computer Marketing Group, Hewlett-Packard Company, Palo Alto, CA. Technical Letter, September 28, 1982.

Appendix A

Microprogramming Concepts

Background

Microprogramming is a lower level of computer programming (Ref. 27:Chapt. 2). Program instructions written in higher level languages (HLL) such as FORTRAN are first translated or compiled into machine-dependent machine language instructions (macroinstructions) in non-real time. Each macroinstruction is then translated or mapped (interpreted) into one or more microinstructions at the time of program execution. This instruction hierarchy is illustrated in Figure 17.

Figure 18 shows an example of a microprogrammed computer architecture. The execution of a program begins with the fetching of the first macroinstruction of the program from main memory. The operation code (opcode) of the macroinstruction points indirectly to the control store (microprogram memory) location of its corresponding microroutine. The microinstructions are sequentially fetched from the control store and executed, activating the various hardware register transfer control points, and ultimately causing the computer to perform the operation specified in the original higher level language instruction. The next macroinstruction is then fetched, and the process continues

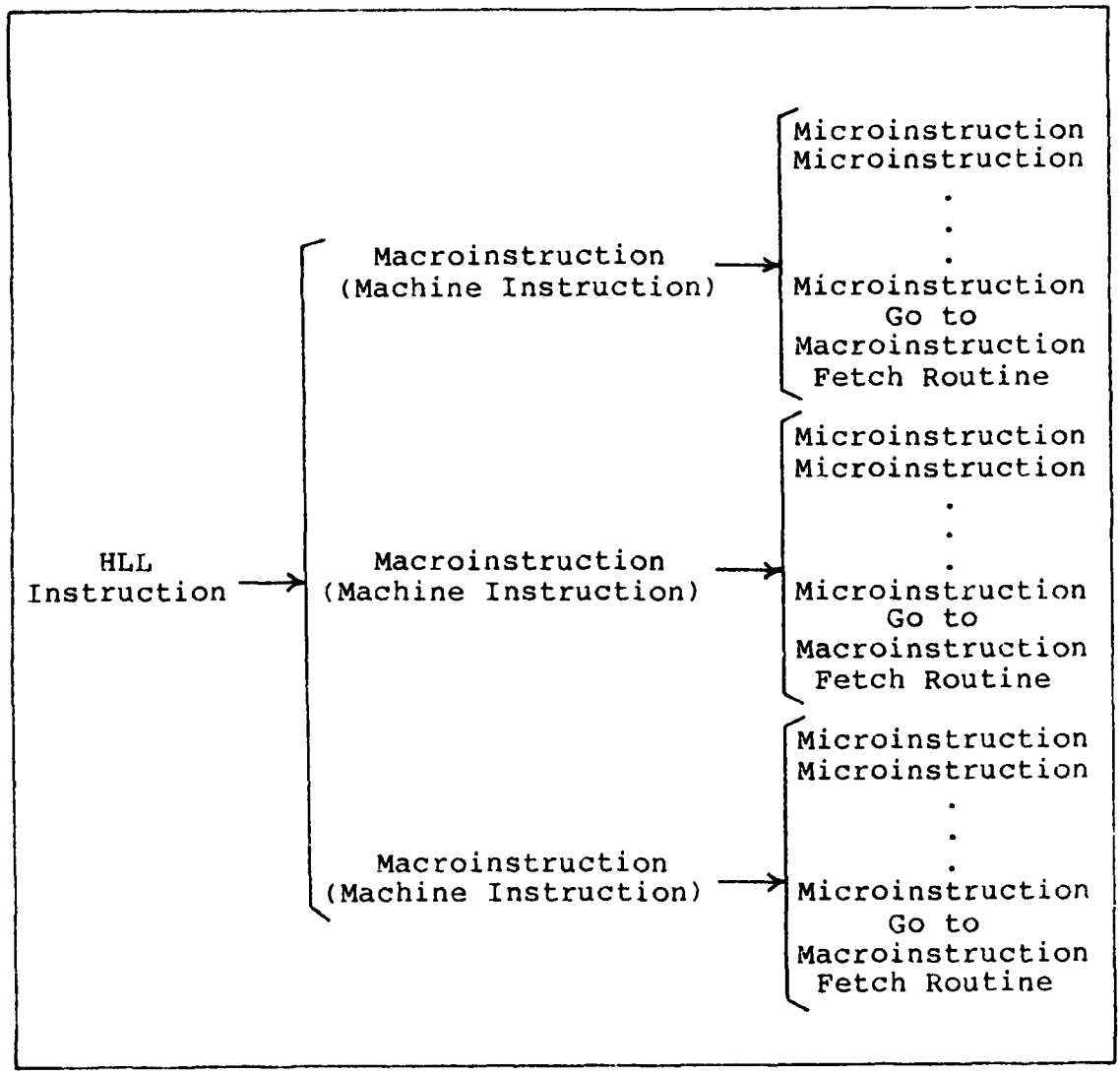


Figure 17. Program Instruction Hierarchy

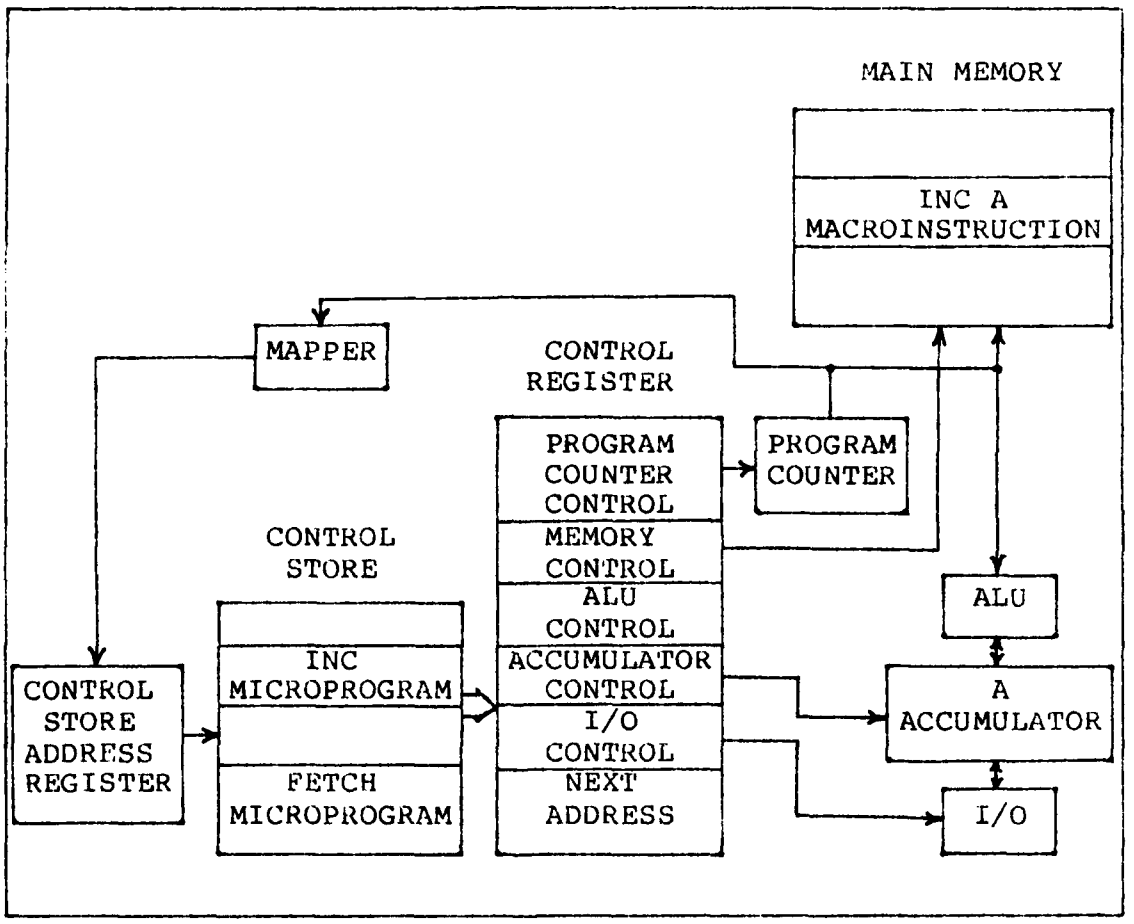


Figure 18. Example Microprogrammed Computer Architecture

until all macroinstructions of the program have been executed. Each macroinstruction is essentially a call to a microroutine. The opcode of the macroinstruction indicates the "name" (address) of the microroutine, and the other fields of the macroinstruction such as address and register fields serve as the parameters to be passed to the microroutine.

Microprogramming requires much greater attention to detail than programming in a higher level language or assembly language, because of the number of lower-level operations and the timing of those operations. The microprogrammer must be concerned with transfers between buses, registers, and main memory, and the operations of the arithmetic logic unit. These transfers and ALU operations are specified in the fields of the microinstruction. These fields are called micro-orders or micro-operations.

Consider, for example, the simple problem of incrementing a variable called A. In a higher level language this can be done with one instruction, such as A=A+1. In assembly language this problem may require three instructions -- an instruction to load the value A into an accumulator register, an instruction to increment the accumulator, and an instruction to store the new value of A back into its memory location. In microcode the problem requires several microinstructions, each microinstruction comprisi several fields or micro-orders. The required micro-orders may be as follows:

1) Move the address of A from the instruction register to a data bus.
2) Move the address from the bus into a memory address register.
3) Read the value A from its memory location into a memory data register.
4) Move the contents of the memory data register to the data bus.
5) Move the value of the data bus to the arithmetic logic unit (ALU).
6) Perform an increment operation in the ALU.
7) Move the result from the ALU back to the data bus.
8) Move the result from the bus back to the memory data register.
9) Write the new value of A back into its memory location.

The number of microinstructions needed to perform these nine micro-orders is dependent on the architecture of the particular machine. Three or four microinstructions is a realistic number. For example, micro-orders 1 through 3 may make up the fields for one microinstruction, 4 through 6 the second, and 7 through 9 the third microinstruction.

How Microprogramming Improves Speed

Since higher level language instructions end up as microinstructions anyway, it is not apparent that directly microprogramming all or part of a program would have any effect on its execution speed. There are hidden factors, however, which do have an effect. Some of the most important factors are instruction fetch time, memory speed, parallelism, and other additional micro-level capabilities.

The time required to fetch an instruction from memory represents 35 to 45% (Ref. 7:11) of the total execution time of an instruction. As shown in Figure 17, each microinstruction of the computer's basic instruction set

concludes with a jump to a macroinstruction fetch routine. By combining the microroutines generated by two or more macroinstructions, the instruction fetches between microroutines are eliminated. Combining microroutines essentially creates a new macroinstruction with the power of several of the original macroinstructions, and only one instruction fetch is required. The elimination of the extra instruction fetches can significantly improve the execution speed.

Microinstructions also must be fetched from memory. The fetch of a microinstruction is, however, significantly faster than that of a macroinstruction. Because the control store is much smaller than a computer's main memory, faster and more expensive memory components can be used. This memory is typically two to five times faster than main memory (Ref. 7:11).

Parallelism is also an important factor in improving execution speed. Since a microinstruction is made up of several microorders, independent parallel operations can be specified in one microinstruction. An example of this concept is the performance of an arithmetic operation and a memory operation at the same time. Parallel operations can provide additional gains in speed. The number of parallel operations is, however, highly dependent on the number of fields in the microword (the width of the microword) and the algorithm being microprogrammed.

The additional capabilities at the micro-level can

also contribute to an increase in execution speed. Additional registers allow the storage of constants, frequently used operands, and intermediate results. This additional storage can often be used to eliminate time-consuming references to main memory. The direct testing of flags and direct shift control can also be used in some applications to improve speed.

Combining all of these factors can provide speed gains many times that of assembly language. Realistic values are between 2 and 20 times (Ref. 9:49).

APPENDIX B

Glossary of Terms Used in this Report (Ref. 7:2)

Arithmetic Logic Unit -- Part of the computer's hardware which performs arithmetic, logic, and other operations.

Assembly Language -- Computer-dependent machine language which is the base instruction set. In a microprogrammed computer, each Assembly language instruction is implemented by a specific microprogram.

Control Processor -- The section of the computer which determines what the computer is to do for each machine instruction.

Control Store -- The memory, used by the Control Processor, in which microprograms reside. It may be implemented with ROM, PROM, and/or WCS.

Fields -- Microinstructions are divided into several parts, known as fields. Each field specifies different micro-operations, which may be independent of one another.

Machine Instructions -- The binary-coded bit patterns that actually control the operations of the computer via

the control Processors. Programs written in symbolic languages, such as FORTRAN, are translated to machine instructions by Compilers, Assemblers, or Interpreters.

Microcode -- Another name for the microinstructions that make up a microprogram, either in source language or in object code form.

Microinstruction -- One instruction of a microprogram, typically made up of one or more micro-orders.

Micro-order -- A complete operation, such as loading a register or setting a register equal to the product of two other registers. Depending upon the control processor, more or less than one micro-order can be specified by a microinstruction.

Microprogram -- A program written for a microprogrammed computer at the control processor level to control the computer. In a totally microprogrammed computer, every machine instruction is implemented by a microprogram.

Microprogramming -- The process of developing microprograms for control of a microprogrammed computer.

PROMs (Programmable Read-Only Memory) and ROMs (Read-Only Memory) -- Are components used to store microprograms in

control store. Once programmed, they cannot be altered. ROMs differ from PROMs in that ROMs have their microprograms installed when they are manufactured while PROMS are programmed after they have been made.

WCS (Writable Control Store) -- Control store implemented with Random Access Memory so that the user can dynamically alter its contents.

APPENDIX C

The following is a list of HP users at Wright-Patterson Air Force Base:

Jim Leonard	AFWAL/AARF-2	55987	Bldg 23
Ken Greer	AFWAL/AARF-2	55987	Bldg 23
Jeff Barnes	AFWAL/AARF-2	55987	Bldg 23
Al Bowling	AFWAL/AARF-2	55987	Bldg 23
Ralph Pinney	AFWAL/AARF-2	55987	Bldg 23
Lloyd Clark	AFWAL/AARF-4	53050	Bldg 23
Glenn Williams	AFWAL/FIMN	52493	Bldg 26/240
Bob Ballard	AFWAL/FIMN	52493	Bldg 26/240
Frank Gondolfi	AFWAL/AAWP	55076	Bldg 821
Bryan Kent	AFWAL/AAWP	55076	Bldg 821
Conrad Phillippi	AFWAL/MLPJ	52334	Bldg 651
John Bankovskis	AFWAL/AARI	56361	Bldg 622
Carl Williams	AFWAL/FIEE	56078	Bldg 45/93
Mike Fabian	AFWAL/FIEE	56078	Bldg 45/93
John Warner	AFWAL/FIEE	56078	Bldg 45/93
Bill Griffin	ASD/ENAMA	55153	Bldg 125
John Steidle	ASD/ENAMA	55153	Bldg 125
Russ Soerens	ASD/ENAMA	55153	Bldg 125
Larry Linder	AFFDL	55205	Bldg 192

Appendix D

This appendix contains listings for the activity profile generator program (ACTV) and its subroutines (RCORE and IDGET). The original program was written by Jim Leonard, AFWAL/AARF-2, WPAFB. The program was modified slightly during this thesis study to run on the AFIT HP 21MX computer.

```
FTN4,L,B
      PROGRAM ACTV
      DIMENSION FBF(52),IPR(52),IN(3)
C     ACTIVITY PROFILE GENERATOR USING SUSPEND ADDRESS
C     FROM ID-SEGMENT.
C        BY JIM LEONARD
C        USAF AVIONICS LAB, WPAFB
C
  5   WRITE(1,10)
 10   FORMAT(" ACTIVITY PROFILE GENERATOR ",//,
     1 " TYPE PROG NAME " )
      IN(1)=2H
      IN(2)=2H
      IN(3)=2H
      READ(1,20) IN
 20   FORMAT(3A2)
C     GET ADDRESS OF ID SEGMENT
      IDSEG=IDGET(IN)
      IF (IDSEG.NE.0) GOTO 100
      WRITE(1,30)IDSEG
 30   FORMAT("IMPROPER PROGRAM NAME, IDSEG= ",I8)
      GOTO 5
 100  WRITE(1,110)
 110  FORMAT("TYPE PROFILE BOUNDS, LOWER-UPPER & INTERRUPT
     1 TIME(1-9)",/," XXXXX XXXXX     X")
      NT=0
      READ(1,120)IL,IU,NT
 120  FORMAT(2K6,I6)
      IF(NT.LE.0)NT=3
C     INITIALIZE PROFILE BUFFER
      DO 130 I=1,52
      FBF(I)=0.
 130  CONTINUE
      ID=IU-IL+1
      INCR=(IU-IL+1)/50
      IF(INCR*50.LT.ID)INCR=INCR+1
      IW1=IDSEG+15
      IW2=IDSEG+8
C    IF PROGRAM IS NOT CURRENTLY ACTIVE DON'T RECORD LOCATION
 300  CALL RCORE(IW1,IVAL)
      IF(IAND(IVAL,15B).NE.1)GOTO 200
C     READ SUSPENDED LOCATION
      CALL RCORE(IW2,IVAL)
C     CHECK FOR BEFORE BOUNDS
      IF (IVAL.GE.IL)GOTO 140
      FBF(1)=FBF(1)+1.
      GOTO 200
C     CHECK FOR BEYOND BOUNDS
 140  IF(IVAL.LE.IU)GOTO 150
      FBF(52)=FBF(52)+1
      GOTO 200
C     MARK INTERVAL
 150  IVAL=(IVAL-IL)/INCR+2
```

```
      FBF(IVAL)=FBF(IVAL)+1.
C     TERMINATE MONITORING IF OPERATOR BREAKS
 200  IF(IFBRK(IDMY))500,210
 210  ISC=0
C     WAIT DESIRED INTERVAL
      CALL EXEC(12,ISC,1,0,-NT)
      GOTO 300
 500  WRITE(10,510)IN,IL,IU,INCR
 510  FORMAT(" PROGRAM ACTIVITY PROFILE FOR ",3A2,/,
     1 "    FROM",K8,"      TO",K8,"    IN INCREMENTS OF",I8)
 515  FORMAT(" INTERVAL NO.  FROM    TO    NO OF HITS    "
     1 ,"NORMALIZED HITS   NORMAL ACCUM")
C     FIND MAX VALUE OF HISTOGRAM
      FMX=-1
      TSUM=0
      DO 520 I=2,51
      TSUM=TSUM+FBF(I)
      IF(FMX.LT.FBF(I))FMX=FBF(I)
 520  CONTINUE
C     EXIT IF NO ACTIVITY IN DESIRED RANGE
      IF(FMX.GT.0)GOTO 600
      IF((FBF(1)+FBF(52)).GT.0)GOTO 540
      WRITE(1,530)
 530  FORMAT("NO PROGRAM ACTIVITY RECORDED--AT ALL!!!")
      WRITE(10,530)
      STOP
 540  WRITE(1,550)FBF(1),FBF(52)
 550  FORMAT("NO PROGRAM ACTIVITY IN REGION OF INTEREST",/
     1  ,"BEFORE=",E13.7,"   AFTER="E13.7)
      WRITE(10,550)FBF(1),FBF(52)
C     WRITE TABLE OF ACTIVITY PROFILE
 600  WRITE(10,515)
      SUM=0.
      TSM1=TSUM+FBF(1)+FBF(52)
      DO 650 I=1,52
      SUM=SUM+FBF(I)/TSM1
      FNORM=FBF(I)/FMX
      IFR=IL+INCR*(I-2)
      ITO=IFR+INCR
      IF(I.EQ.1)IFR=0
      IF(I.EQ.52)ITO=32767
      WRITE(10,610)I,IFR,ITO,FBF(I),FNORM,SUM
 610  FORMAT(4X,I3,6X,2K7,F10.0,F17.8,F15.5)
 650  CONTINUE
C     PLOT HISTOGRAM ON PRINTER
      WRITE(10,510)IN,IL,IU,INCR
      WRITE(10,700)
 700  FORMAT(" INTERVAL  0         2         4         6"
     1  "         8         1")
C     FOR EACH DATA INTERVAL
      SUM=-FBF(1)/TSUM
      DO 800 J=1,52
C     CLEAR PRINTER BUFFER
```

```
      DO 710 I=1,51
      IPR(I)=2H
  710 CONTINUE
C     CALCULATE INDEXS
      SUM=SUM+FBF(J)/TSUM
      INDX=SUM*50.+1.5
      IF((J.NE.1).AND.(J.NE.52))IPR(INDX)=2HII
      INORM=50.*FBF(J)/FMX+1.5
C     PRINT AN X IF OFF PLOT
      IF(INORM.LT.1)INORM=-1
      IF(INORM.GT.51)INORM=-51
C     PRINT AN * IF ON THE PLOT
      IF(INORM.GT.0)IPR(INORM)=2H00
      IF(INORM.LT.0)IPR(-INORM)=2HXX
      WRITE(10,720)J,(IPR(K),K=1,51)
  720 FORMAT(2X,I6,3X,51A1)
  800 CONTINUE
      STOP
      END
      END$
```

```
ASMB,L
      NAM RCORE
*
*     READS AND RETURNS THE CONTENTS OF A SINGLE
*     MEMORY LOCATION.
*
*     THIS IS A SUBROUTINE TO THE ACTIVITY PROFILE
*     GENERATOR JIM LEONARD WROTE.
*
*    THE ACTIVITY PROFILE SOURCE PROGRAM IS ON FILE &ACTV::20
*
*     JOHN STEIDLE
*
      ENT RCORE
      EXT .ENTR
IW1   NOP             ADDRESS OF ADDRESSES OF DESIRED VALUE
IW2   NOP             ADDRESS FOR RETURNED CORE VALUE
RCORE NOP
      JSB .ENTR       GET PARAMETER ADDRESSES
      DEF IW1
      LDA IW1,I       READ ADDRESS
      LDA 0,I         READ CONTENTS
      STA IW2,I       STORE IT
      JMP RCORE,I
      END
```

```
FTN4,L
      INTEGER FUNCTION IDGET(IN)
      DIMENSION IN(3)
C FUNCTION IDGET FINDS THE ADDRESS OF THE ID SEGMENT
C OF THE PROGRAM NAME PASSED BY THE CHARACTER ARRAY
C "IN".  THIS FUNCTION IS PERFORMED BY SEQUENTIALLY
C SEARCHING THROUGH THE ID SEGMENTS OF THE SYSTEM
C LOOKING FOR A MATCH ON THE INPUT PROGRAM NAME.  WHEN
C THE CORRECT ID SEGMENT IS FOUND, THE ADDRESS OF THE
C SEGMENT IS PASSED BACK IN "IDGET".  IF THE SEGMENT
C IS NOT FOUND, "IDGET" IS SET TO ZERO.
C
C
C GET ADDRESS OF ID SEGMENT ADDRESS TABLE IN LOC 1657 OCTAL
      IPTR1=1657B
      CALL RCORE(IPTR1,IPTR2)
C LOOP TO SEARCH THROUGH ID SEGMENT TABLES
 900  CALL RCORE(IPTR2,IDGET)
      IF (IDGET .EQ. 0) GOTO 950
      IPTR2=IPTR2+1
C POINT TO NAME AREA OF TABLE AND COMPARE THE 3 WORDS
C CONTAINING THE 5 CHARACTER PROGRAM NAME
      IPTR1=IDGET+12
      CALL RCORE(IPTR1,INAME)
      IF (INAME .NE. IN(1)) GOTO 900
      IPTR1=IPTR1+1
      CALL RCORE(IPTR1,INAME)
      IF (INAME .NE. IN(2)) GOTO 900
      IPTR1=IPTR1+1
      CALL RCORE(IPTR1,INAME)
C COMPARE 5TH CHAR IN UPPER BYTE OF THE WORD.
C IGNORE THE LOWER BYTE.
      IF (IABS(INAME-IN(3)) .GT. 255) GOTO 900
 950  RETURN
      END
      END$
```

Appendix E

The following are instructions for running ACTV on the AFIT RTE-III system:

1. ACTV and the program to be tested must be compiled and loaded prior to running ACTV. If the core image files already exist (saved from a previous session), type the following commands:

```
RP,ACTV
RP,test program name
EX
Any key to get the * prompt
```

If the relocatable object files (the % files) for ACTV or the test program have been loaded during the current session, the corresponding RP command can be ommitted.

2. Set the priority of ACTV to 89 by the following command:

```
PR,ACTV,89
Any key to get the * prompt
```

3. Run ACTV by typing:

```
RU,ACTV
```

4. ACTV responds:

```
ACTIVITY PROFILE GENERATOR
TYPE PROG NAME
```

5. Enter the 5-character program name.

6. ACTV responds:

TYPE PROFILE BOUNDS, LOWER-UPPER & INTERRUPT

XXXXX XXXXX X TIME (1-9)

7. Enter the octal address bounds of the program region ACTV is to monitor and the rate at which the program is to be interrupted. The smaller the interrupt time number, the greater the interrupt rate and total "hits" for the profile. Both the addresses and interrupt rate must be entered directly below the Xs.

8. Press any key to get the * prompt. Run the program under test by:

RU,program name

9. The program will execute normally. When it terminates, type:

BR,ACTV

This will terminate ACTV, and the activity profile will be printed on the printer.

Appendix F

This appendix contains listings for SDRVR, STRES, and SPEED. SDRVR is a special driver program, which was written by AFWAL/FIMN personnel, to test the wind tunnel routines SDRVR and SPEED on the AFIT 21MX computer. SDRVR and SPEED are the original subroutines used in the wind tunnel control program. The routines here are presented essentially as they were received from the user. They are not well commented, and no attempt was made to improve this.

```
FTN4,L
C     12 JUL 82
      PROGRAM SDRVR
C
C
C
      COMMON DDFL(13,13),IZTHM(1),CON1(1),CON6(1)
      DIMENSION IBXNPS(10,18),IBXJAK(8),
     *STRESS(14),YZT(14),YZT1(10)
      REAL MOM(14)
      DIMENSION DFL(13,13),DFL2(13,4),DFL3(13,3)
C
      EQUIVALENCE (DFL(1,7),DFL2),(DFL(1,11),DFL3),
     1            (YZT1,YZT(5))
C     DATA DFL/
     1 .2528E4,-.1750E4,.5417E3,-.4790E2,.1233E2,
     1 -.2884E1,.7545E0,-.2028E0,.5681E-1,-.1703E-1,
     1 .4409E-2,-.1102E-2,.1837E-3,
     2 -.1750E4,.1968E4,-.9478E3,.1805E3,-.4645E2,
     2 .1087E2,-.2843E1,.7644E0,-.2141E0,.6416E-1,
     2 -.1662E-1,.4154E-2,.6923E-3,
     3 .5417E3,-.9478E3,.7798E3,-.3674E3,.1297E3,
     3 -.3035E2,.7941E1,-.2135E1,.5979E0,-.1792E0,
     3 .4640E-1,-.1160E-1,.1933E-2,
     4 -.4790E2,.1805E3,-.3674E3,.4015E3,-.2481E3,
     4 .9002E2,-.2355E2,.6331E1,-.1773E1,.5315E0,
     4 -.1376E0,.3441E-1,-.5734E-2,
     5 .1233E2,-.4646E2,.1297E3,-.2481E3,.2719E3,
     5 -.1828E3,.8358E2,-.2247E2,.6293E1,-.1886E1,
     5 .4884E0,-.1221E0,.2035E-1,
     6 -.2886E1,.1087E2,-.3035E2,.9002E2,-.1828E3,
     6 .3075E3,-.2823E3,.1136E3,-.3181E2,.9533E1,
     6 -.2469E1,.6172E0,-.1029E0/
      DATA DFL2/
     7 .7568E0,-.2845E1,.7942E1,-.2355E2,.8358E2,
     7 -.2823E3,.3976E3,-.2682E3,.1144E3,-.3427E2,
     7 .8876E1,-.2219E1,.3698E0,
     8 -.2045E0,.7653E0,-.2135E1,.6331E1,-.2247E2,
     8 .1136E3,-.2682E3,.3449E3,-.2705E3,.1231E3,
     8 -.3187E2,.7968E1,-.1328E1,
     9 .5761E-1,-.2149E0,.5976E0,-.1772E1,.6292E1,
     9 -.3181E2,.1144E3,-.2705E3,.4069E3,-.3179E3,
     9 .1186E3,-.2965E2,.4942E1,
     * -.1772E-1,.6570E-1,-.1800E0,.5307E0,-.1885E1,
     * .9532E1,-.3427E2 ,.1231E3,-.3179E3,.4359E3,
     * -.3609E3,.1752E3,-.2921E2/
      DATA DFL3/
     1 .5690E-2,-.1935E-1,.4937E-1,-.1388E0,.4870E0,
     1 -.2465E1,.8871E1 ,-.3187E2,.1186E3,-.3609E3,
     1 .6241E3,-.4961E3,.1394E3,
     2 -.2199E-2,.6440E-2,-.1477E-1,.3634E-1,-.1211E0
     2 .6138E0,-.2215E1,.7965E1,-.2965E2,.1752E3,
     2 -.4961E3,.5492E3,-.2049E3
```

```
     3 .4995E-3,-.1367E-2,.3053E-2,-.6550E-2,.2009E-1
     3 -.1017E0,.3685E0,-.1327E1,.4941E1,-.2921E2,
     3 .1394E3,-.2049E3,.9083E2/
C
C
C  LINES ADDED TO REPLACE READS OF DEVICES NOT AVAILABLE
C
      DATA IBXNPS/40*0,10*2423,2024,2064,2137,2234,2302,
     * 2153,2021,1850,1686,1524/
      IBXJAK(1) = 2172
      IBXJAK(2) = 2365
      IBXJAK(3) = 2422
      IBXJAK(4) = 2000
      IBXJAK(5) = 2000
      IBXJAK(6) = 2000
      IBXJAK(7) = 0
      IBXJAK(8) = 0
      DO 35 I=1,169
         DDFL(I) = DFL(I)
 35   CONTINUE
C
C
      IZTHM=2000
      CON1=5E-4
      CON6=5.4932E-4
C
      DO 2000 IROD=5,6
         CALL STRES(1,IROD,IBXJAK,IBXNPS(1,IROD),STRESS,MOM)
         IBXJAK(1) = 2000
         IBXJAK(2) = 2000
         IBXJAK(3) = 2000
2000  CONTINUE
      END
```

```
C
C
      SUBROUTINE STRES(IFCN,KIROD,IBXJAK,IDATA,STRESS,MOM,
     *                 IZXNPP)
C
C
C  FCN=1 FOR THUMWHEEL
C  FCN=2 FOR POTS
C  LAST PARAMETER IS NOT REQUIRED FOR THUMWHEELS
C
C
      COMMON DDFL(13,13),IZTHM(1),CON1(1),CON6(1)
      DIMENSION IBXJAK(8),STRESS(14),YZT(14),YZT1(10),
     *IDATA(10),IBXNPP(10)
      REAL JACK(14),LOAD(14),MOM(14)
      EQUIVALENCE (YZT1,YZT(5))
      DATA JACK/0.,6.,11.,16.,21.,26.,31.,33.5,36.,38.5,41.,
     *44.69,48.38,52.07/
      YZT(1)=0
      IF (IFCN.EQ.3) GO TO 3000
      IF (KIROD.GT.9) GO TO 100
      YZT(2)= (IBXJAK(1)-IZTHM)* CON1
      YZT(3)= (IBXJAK(2)-IZTHM)* CON1
      YZT(4)= (IBXJAK(3)-IZTHM)* CON1
      GO TO 200
  100 YZT(2)= (IBXJAK(4)-IZTHM)* CON1
      YZT(3)= (IBXJAK(5)-IZTHM)* CON1
      YZT(4)= (IBXJAK(6)-IZTHM)* CON1
  200 CONTINUE
      IF (IFCN.EQ.2) GO TO 2000
 1000 DO 1050 I=1,10
 1050    YZT1(I)= (IDATA(I)-IZTHM)* CON1
 1500    CALL SPEED(LOAD,MOM,STRESS,YZT,JACK)
      RETURN
 2000 DO 2050 I=1,10
 2050 YZT1(I)= (IDATA(I)-IZXNPP(I))* CON6
      CALL SPEED(LOAD,MOM,STRESS,YZT,JACK)
      RETURN
 3000 CALL SPEED(LOAD,MOM,STRESS,YZT,JACK)
      RETURN
      END
      END$
```

```
ASMB,L
       NAM SPEED,7
       EXT .ENTR
       ENT SPEED
       COM DDFL(338)
LOAD   BSS 1
MOM    BSS 1
STRES  BSS 1
YZT    BSS 1
JACK   BSS 1
SPEED  NOP
       JSB .ENTR
       DEF LOAD
       LDA LOAD
       INA
       INA
       STA .LOD1
       STA .LOD2
       STA .LOD3
       STA .LOD4
       LDA YZT
       INA
       INA
       STA .YZT
       LDA JACK
       INA
       INA
       STA .JCK1
       STA .JCK2
       STA .JCK4
       LDA MOM
       STA .MOM1
       STA ..A
       INA
       INA
       STA .MOM3
       LDA ..DFL
       STA .DDFL
*      COMPUTE RESULTS BASED ON DEFLECTIONS ALONE
*      FIND LOADS
       LDA =D-13
       STA CNT2
LOOP2  LDA .YZT
       STA ..YZT
       DLD .DDFL,I
       OCT 105040     FMP
.YZT   BSS 1
       DST .LOD1,I
       ISZ ..YZT
       ISZ ..YZT
       ISZ .DDFL
       ISZ .DDFL
       LDA =D-12
       STA CNT1
```

```
LOOP1 DLD .DDFL,I
      OCT 105040     FMP
..YZT BSS 1
      OCT 10500      FAD
.LOD1 BSS 1
      DST .LOD1,I
      ISZ ..YZT
      ISZ ..YZT
      ISZ .DDFL
      ISZ .DDFL
      ISZ CNT1
      JMP LOOP1
      ISZ .LOD1
      ISZ .LOD1
      ISZ CNT2
      JMP LOOP2
*     FIND MOMENT DISTRIBUTION AT JACKS
      LDA =D-13
      STA CNT1
      CLA
      CLB
LOOP3 OCT 105020     FSB
.LOD2 BSS 1
      ISZ .LOD2
      ISZ .LOD2
      ISZ CNT1
      JMP LOOP3
      DST LOAD,I
      CLA
      CLB
      DST .MOM1.I
      LDA =D-13
      STA CNT2
LOOP4 DLD .JCKA,I
      OCT 105040     FMP
.LOD3 BSS 1
      OCT 105000     FAD
.MOM1 BSS 1
      DST .MOM1,I
      ISZ .JCK1
      ISZ .JCK1
      ISZ .LOD3
      ISZ .LOD3
      ISZ CNT2
      JMP LOOP4
      DLD LOAD,I
      DST TEMP1
      DLD MOM,I
      DST TEMP2
      LDA =D-12
      STA CNT1
LOOP5 DLD TEMP1
      OCT 10500      FAD
.LOD4 BSS 1
```

```
        DST TEMP1
        DLD .LOD4,I
        OCT 105040      FMP
.JCK4   BSS 1
        DST TEMP3
        DLD TEMP2
        OCT 105020      FSB
        DEF TEMP3
        DST TEMP2
        DLD .JCK4,I
        OCT 105040      FMP
        DEF TEMP1
        OCT 105000      FAD
        DEF TEMP2
        DST .MOM3,I
        ISZ .MOM3
        ISZ .MOM3
        ISZ .LOD4
        ISZ .LOD4
        ISZ .JCK4
        ISZ .JCK4
        ISZ CNT1
        JMP LOOP5
*       STRESS AT JACK CENTERLINE AND WALL
        LDA .MP
        STA ..M
        LDA STRES
        STA ..S
        LDA =D-13
        STA CNT2
LOOP7   DLD ..A,I
        ISZ ..A
        ISZ ..A
        SSA
        CMA,INA
        OCT 105040      FMP
..M     BSS 1
        DST ..S,I
        ISZ ..S
        ISZ ..S
        ISZ ..M
        ISZ ..M
        ISZ CNT2
        JMP LOOP7
        JMP SPEED,I
..DFL   BSS 1
.JCK1   BSS 1
.JCK2   BSS 1
.JCK3   BSS 1
.MOM3   BSS 1
CNT1    BSS 1
CNT2    BSS 1
..A     BSS 1
..S     BSS 1
```

```
.MP     DEF MP
MP      DEC 148.63,148.63,148.63
        DEC 594.5,594.5,594.5
        DEC 2378.,2378.,2378.,2378.
        DEC 2378.,594.5,594.5
TEMP1   BSS 2
TEMP2   BSS 2
TEMP3   BSS 2
        END SPEED
```

Appendix G

This appendix contains listings for MSPED and LOADS. MSPED is a version of SPEED modified to invoke a microprogram substitution for LOOP1 and LOOP2 of SPEED. Only the code up to and including the modifications are shown here. The remainder of the code is as in the SPEED listing of Appendix F. Also, the program name remains as "SPEED" to minimize changes to calling routines. Only the file names are changed to MSPED. LOADS is the microprogram subsitution for LOOP1 and LOOP2.

```
ASMB,L
       NAM SPEED,7
       EXT .ENTR
       ENT SPEED
       COM DDFL(338)
LOAD   BSS 1
MOM    BSS 1
STRES  BSS 1
YZT    BSS 1
JACK   BSS 1
SPEED  NOP
       JSB .ENTR
       DEF LOAD
       LDA LOAD
       INA
       INA
       STA .LOD1
       STA .LOD2
       STA .LOD3
       STA .LOD4
       LDA YZT
       INA
       INA
       STA .YZT
       LDA JACK
       INA
       INA
       STA .JCK1
       STA .JCK2
       STA .JCK4
       LDA MOM
       STA .MOM1
       STA ..A
       INA
       INA
       STA .MOM3
       LDA ..DFL
       STA .DDFL
*      COMPUTE RESULTS BASED ON DEFLECTIONS ALONE
*      FIND LOADS BY INVOKING THE LOADS MICROPROGRAM
LOADS  OCT 105600
.DDFL  BSS 1
.YZT   BSS 1
.LOD1  BSS 1
*      FIND MOMENT DISTRIBUTION AT JACKS
        .
        .    THE CODE HERE IS IDENTICAL TO SPEED
        .
       END SPEED
```

```
MICMX,L,R                                   21MX
$CODE=%LOADS::20,REPLACE                    OBJECT TO DISK
         ORG  6000B
*************************************************************
*                                                           *
*                    LOADS  MICROPROGRAM                    *
*                                                           *
* THIS MICROPROGRAM IS A SUBSTITUTE FOR THE ASSEMBLY        *
* LANGUAGE CODE SEGMENT LABELED LOOP2 IN THE ROUTINE        *
* CALLED SPEED. WITH THIS ROUTINE WRITTEN INTO WCS,         *
* THE LOOP2 CODE SEGMENT IN SPEED (FROM THE LABEL           *
* "LOOP2" TO THE "JMP LOOP2" INSTRUCTION) CAN BE            *
* REPLACED BY THE FOLLOWING INSTRUCTIONS:                   *
*                                                           *
* LOADS OCT 105600    CALL THE LOADS MICROPROGRAM           *
* .DDFL BSS 1         ADDRESS OF THE DDFL ARRAY             *
* .YZT  BSS 1         ADDRESS OF THE YZT ARRAY              *
* .LOD1 BSS 1         ADDRESS OF THE LOAD ARRAY             *
*                                                           *
*NOTE THAT .DDFL, .YZT, AND .LOD1 ARE ALREADY DEFINED       *
*IN SPEED. THEY MUST BE MOVED TO THE LINES FOLLOWING        *
*THE "LOADS OCT 105600" INSTRUCTION AS SHOWN ABOVE.         *
*THE ORDER IS IMPORTANT AS THESE ARE PARAMETERS FOR         *
*THE MICROPROGRAM.                                         *
*                                                           *
*************************************************************

FLD      EQU                  %7031   ROM FLT PNT LOAD ROUTINE
PACK     EQU                  %7052   ROM FLT PNT PACK ROUTINE

START    JMP                  LOADS   JUMP TO MAIN MICROPROGRAM
         ORG  6002B                   USE 6001B FOR DEBUG BKPNT

*************************************************************
* READ CALLING PARAMETERS FROM MEMORY AND STORE IN          *
* SCRATCH REGISTERS:     .DDFL --> S3                       *
*                        .YZT  --> S12                      *
*                        .LOD1 --> S8                       *
* ALSO INITIALIZE OUTER LOOP COUNTER REGISTER X TO 13       *
*************************************************************
LOADS    READ       INC  M    P      READ DDFL ADDR FROM MEMORY
                    INC  P    P      POINT TO YZT ADDRESS
                    PASS S3   TAB    PUT DDFL ADDRESS INTO S3
         READ       INC  M    P      READ YZT ADDR FROM MEMORY
                    INC  P    P      POINT TO LOAD ARRAY ADDR
                    PASS S12  TAB    PUT YZT ADDRESS INTO S12
         READ       INC  M    P      READ LOAD ADDR FROM MEMORY
         IMM        CMLO X    %362   LOOP2 CNTR=13(1'S CMP 362)
                    PASS S8   TAB    PUT LOAD ADDRESS INTO S8

*************************************************************
* MATRIX MULTIPLICATION LOOP -- THIS CODE SEGMENT           *
* PERFORMS THE FLOATING POINT MATRIX MULTIPLICATION         *
* OF THE 13X13 DDFL MATRIX BY THE 14X1 YZT MATRIX.          *
```

```
* THE FIRST ELEMENT OF THE YZT MATRIX IS NOT USED,    *
* MAKING IT EFFECTIVELY A 13X1 MATRIX. THE RESULT OF *
* THE MATRIX MULTIPLICATION IS THE 14X1 MATRIX CALLED*
* LOAD (1ST ELEMENT AGAIN NOT USED).                 *
*****************************************************

LOOP2     IMM        CMLO A    %377    PUT A ZERO IN A-REG
               MPCK  INC  M    S8      PUT LOAD ADDR INTO M-REG
          WRTE       PASS TAB  A       CLEAR WORD1 OF LOAD ELEMNT
                     INC  B    S8      POINT B TO WORD2 OF LOAD
               MPCK  INC  M    B       & PUT ADDR INTO M-REG
          WRTE       PASS TAB  A       CLEAR WORD2 OF LOAD ELEMNT
          IMM        CMLO Y    %362    LOOP1 CNTR=13(1'S CMP 362)
                     PASS S4   S12     PUT YZT ADDR INTO S4
LOOP1     READ       INC  M    S3      READ 1ST WORD DDFL ELEMENT
                     INC  S3   S3      POINT TO 2ND DDFL WORD
                     PASS A    TAB     PUT 1ST WORD INTO A-REG
          READ       INC  M    S3      READ 2ND WORD OF DDFL
                     INC  S3   S3      POINT TO NEXT DDFL ELEMENT
                     PASS B    TAB     PUT 2ND WORD INTO B-REG
*****************************************************
* "JMP" RATHER THAN "JSB" TO FMPY AND FADD ROUTINES  *
* BECAUSE THESE ROUTINES WILL DESTROY TH RETURN      *
* ADDRESS BY CALLING OTHER ROUTINES. RETURN IS TO THE*
* INSTRUCTIONS LABELED "RTNFMPY" AND "RTNFADD"       *
*****************************************************

          JMP                  FMPY    MULTIPLY DDFL&YZT ELEMENTS
*                                       PRODUCT GOES INTO A/B REGS
RTNFMPY   JMP                  FADD    ADD DDFL&YZT PROD TO LOAD
*                                       SUM GOES INTO A/B REGS
RTNFADD        MPCK INC  M    S8      PUT LOAD ADDR INTO M-REG
          WRTE       PASS TAB  A       WRITE A-REG TO LOAD ADDR
                     INC  S8   S8      POINT TO 2ND WORD OF LOAD
               MPCK  INC  M    S8      PUT ADDRESS INTO M-REG
          WRTE       PASS TAB  B       WRITE B-REG TO 2ND WORD
                     DEC  S8   S8      POINT BACK TO 1ST WORD
                     INC  S4   S4      POINT TO NEXT YZT WORD
                     INC  S4   S4
                     DEC  Y    Y       DECREMENT LOOP1 COUNTER
          JMP  CNDX  TBZ  RJS  LOOP1   IF CNTR NOT=0 GO TO LOOP1
                     INC  S8   S8      POINT TO NEXT LOAD ELEMENT
                     INC  S8   S8
                     DEC  X    X       DECREMENT LOOP2 COUNTER
          JMP  CNDX  TBZ  RJS  LOOP2   IF CNTR NOT=0 GO TO LOOP2
               RTN   INC  P    P       RETURN TO SPEED
*****************************************************
* FLOATING POINT MULTIPLY AND MULTIPLY (MPYX)        *
* ROUTINES. THESE ROUTINES ARE TAKEN FROM APPENDIX E *
* OF THE HP MICROPROGRAMMING 21MX COMPUTERS OPERATING*
* AND REFERENCE MANUAL. THESE ROUTINES ALSO RESIDE IN*
* CONTROL STORE ROM, BUT IT IS NECESSARY TO REPRODUCE*
* THEM IN WCS TO AVOID THE PROBLEM OF LEVELED SUB-   *
* ROUTINE CALLS IN THE M-SERIES. FMPY HAS BEEN MODI- *
```

```
* FIED SLIGHTLY TO HANDLE A PARAMETER ADDRESS IN A    *
* SCRATCH REGISTER RATHER THAN POINTED AT BY THE P     *
* REGISTER (PROGRAM COUNTER). ALSO, INDIRECT ADDRES-   *
* SING IS NOT USED. MPYX IS UNCHANGED.                 *
*******************************************************

FMPY    READ        INC   M    S4      READ 1ST PARAMETER WORD
        JSB                    FLD     STORE ARGS IN SCRATCH REGS
                    INC   S9   S9
                    PASS  L    S5      FORM EXP1+EXP2+1
                    ADD   S9   S9      AND SAVE IN S9
        R1          PASS  A    S10     FORM (WORD1 LOBITS)/2 IN A
                    PASS  S2   S7      PASS WORD2 HIBITS INTO S2
        JSB                    MPYX    JMP TO MPY SUB & RTN WITH
                    PASS  S5   B       HIBITS IN B & SAVE IN S5
                    PASS  S2   S11     PASS WORD1 HIBITS INTO S2
                    PASS  S11  A       LOBITS INTO A. SAVE INTO S
              R1    PASS  A    S6      FORM (WORD2 LOBITS)/2 IN A
        JSB                    MPYX    JMP TO MPY SUB & RTN WITH
                    PASS  L    A       LOBITS IN A & PASS INTO L
                    ADD   A    S11     ADD BOTH LOBITS. CHK FOR C
        JMP   CNDX  COUT  RJS  *+2     (ELSE TRUNCATE DIGITS)
                    INC   B    B       IF COUT, BUMP HIBITS
                    PASS  L    B       ADD HIBITS & SAVE IN S11
                    ADD   S11  S5
                    PASS  A    S7      PASS WORD2 HIBITS INTO A
        JSB                    MPYX    JMP TO MPY SUB & RTN WITH
              R1    PASS  A    A       LOBITS IN A. SAVE LOBITS/2
              COV   PASS  L    A       ADD LOBITS/2 TO HIBITS SUM
        ENV   L1    ADD   A    S11     SHIFT L1 TO REORIENT
        JMP   CNDX  AL15  RJS  *+3     CHECK FOR CARRY INTO OR
        JMP   CNDX  OVFL       *+4     BORROW FROM HIBITS &
                    DEC   B    B       ADJUST ACCORDINGLY
        JSB                    PACK
        JMP                    RTNFMPY RTN TO MAIN MICROPROGRAM
                    INC   B    B       CAN'T OVERFLOW FROM HIBITS
        JSB                    PACK
        JMP                    RTNFMPY RTN TO MAIN MICROPROGRAM

MPYX          COV   PASS  S1   A       S1<-A(MULTIPLICAND). CLEAR
                    ZERO  B            CLEAR B FOR MULTIPLY
                    PASS  L    S2      L<-S2(MULTIPLIER)
              RPT   PASS  CNTR B       CLEAR COUNTER & SET REPEAT
        MPY   R1    ADD   B    B       MPY STEP (X16), (B,A)<-*L+
                    PASS       S1      TEST MULTIPLICAND
        JMP   CNDX  AL15  RJS  *+2     JUMP IF POSITIVE
                    SUB   B    B       UNDO LAST MPY STEP IF NEG
                    PASS       S2      TEST MULTIPLIER
        JMP   CNDX  AL15  RJS  RETURN  JMP IF POSITIVE
                    PASS  L    S1      L<-MULTIPLICAND
              RTN   SUB   B    B       B<-MINUS L (CORRECTS NEG
*                                      MULT)
RETURN        RTN                      RETURN TO CALLING ROUTINE
```

```
*********************************************************
* FLOATING POINT ADD ROUTINE. THIS ROUTINE IS ALSO      *
* TAKEN FROM APPENDIX E OF THE HP MICROPROGRAMMING      *
* 21MX COMPUTERS OPERATING AND REFERENCE MANUAL. IT     *
* ALSO RESIDES IN CONTROL STORE ROM BUT IS DUPLICATED   *
* IN WCS TO AVOID THE PROBLEM OF LEVELED SUBROUTINE     *
* CALLS IN THE M-SERIES. FADD HAS BEEN MODIFIED TO      *
* EXCLUDE THE CODE FOR FLOATING POINT SUBTRACT AND TO   *
* ALLOW A PARAMETER ADDRESS IN A SCRATCH REGISTER       *
* RATHER THAN THE P REGISTER. INDIRECT ADDRESSING IS    *
* NOT USED, SO THE CALL TO "INDIRECT" IS OMITTED.       *
* ALSO, SCRATCH REGISTER S2 IS USED IN PLACE OF S8 TO   *
* FREE S8 FOR USE IN THE MAIN PROGRAM.                  *
*********************************************************

FADD     READ        INC   M    S8      READ 1ST PARAMETER WORD
         JSB                    FLD     UNPACK WORDS INTO SCR REGS
                     PASS  B    S7      CHECK FOR WORD2=0
         JMP  CNDX   TBZ   RJS  *+2     IF NOT, CONTINUE
         IMM         LOW   S5   %200    IF SO, MAKE EXP MOST NEG
                     PASS       S11     CHECK FOR WORD1=0
         JMP  CNDX   TBZ   RJS  *+2     IF NOT, CONTINUE
         IMM         LOW   S9   %200    IF SO, MAKE EXP MOST NEG
DIFR                 PASS  A    S6
                     PAS   L    S5      FIND DIFF IN EXPS
              CLFL   SUB   S2   S9      & STORE IN S2, FLAG=0
         JMP  CNDX   TBZ        ADD2    IF DIFF=0, JMP TO ADD STEP
         JMP  CNDX   AL15       RVRS    IF NEG, WORD2>WORD1
                     CMPS  S2   S2      FORM -DIFF
                     INC   S2   S2      & STORE -DIFF IN S2
         JMP                    SWAMPCHK
RVRS                 PASS  L    B       HOLD B IN L
                     PASS  B    S11     WORD1<WORD2, FILL IN B,A
                     PASS  A    S10     WITH S11,S10
                     PASL  S11          ALSO FILL S11,S10,S9
                     PASS  S10  S6      WITH B,S6,S5
                     PASS  S9   S5
SWAMPCHK IMM         LOW   L    %350    FORM -30B8 IN L
                     SUB        S2      IF -DIFF>-31,RTN WITH
*                                       LARGER #
         JMP  CNDX   AL15       OUT     JMP TO RESTORE A,B
SHIFT    ARS  R1     PASS  B    B       NOW START SHIFT LOOP
                     INC   S2   S2      INC COUNTER
         JMP  CNDX   TBZ   RJS  SHIFT   LOOP UNTIL DONE
ADD2          COV    PASS  L    S10     PASS LOBITS INTO L
                     ADD   A    A       ADD & CHECK FOR COUT
         JMP  CNDX   COUT  RJS  *+3     IF NOT, JUMP
         IMM         HIGH  L    %0      CLR L(15) FOR OVFL
         ENV         INC   B    B       IF SO,INC HIBITS & ENABLE
*                                       OVERFLOW
              CLFL   PASS  L    S11     FLAG=0
         ENV         ADD   B    B       ADD HIBITS & ENABLE OVFL
         JMP  CNDX   OVFL  RJS  PKSUB   IF NO OVFL, RETURN
         JMP  CNDX   AL15       *+2     OVFL IMPLIES SIGN CHANGE
```

```
                STFL                            SO FLAG=U IF AL15=0
          LWF   R1    PASS B    B               DO FULL WORD SHIFT
          LWF   R1    PASS A    A               USING FLAG REG TO INJECT
*                                                 SIGN
                      INC  S9   S9              BUMP EXP
PKSUB     JSB                   PACK            REPACK A,B REGS
          JMP                   RTNFADD  RTN TO MAIN MICROPROGRAM
OUT                   PASS B    S11             PASS MUCH LARGER WORD INTO
*                                                 B,A
                      PASS A    S10
          JSB                   PACK
          JMP                   RTNFADD  RTN TO MAIN MICROPROGRAM
          END
```

Appendix H

This appendix contains the listing for WCSLD. This program was used to load the LOADS microprogram into WCS on the AFWAL/FIMN HP 21MX. The microprogram object code is predefined in a buffer, and the buffer is output to the WCS two words at a time. The program was run under a DOS III operating system which had not been configured for microprogramming. The program will not run on the AFIT RTE III system because of the installed memory protect option. The direct I/O instructions (STF, STC, OTA, OTB, LIA, LIB) cause memory protect violations.

```
ASMB,L
        NAM WCSLD,3
        ENT WCSLD
*
*
* WCSLD WRITES A BUFFER CONTAINING PRESTORED MICROCODE OUT
* TO WCS. ONCE THE MICROCODE IS WRITTEN TO WCS, THE PROGRAM
* READS THE MICROCODE FROM WCS, COMPARES IT WITH THE CODE
* THAT WAS OUTPUT, AND WRITES THE INPUT CODE INTO ANOTHER
* BUFFER. IF A WORD DOES NOT COMPARE, AN ERROR COUNTER IS
* INCREMENTED. THE MICROCODE IS WRITTEN OUT 2 WORDS AT A
* TIME. THE UPPER 8 BITS OF THE A-REG CONTAINS THE WCS
* ADDRESS (0-377B8), AND THE LOWER 8 BITS OF THE A-REG
* CONTAINS THE UPPER 8 BITS OF THE MICROWORD. READING IS ALSO
* DONE 2 WORDS AT A TIME. THE WCS ADDRESS IS FIRST OUTPUT TO
* THE BOARD, AND THEN THE MICROWORD AT THAT ADDRESS IS READ
* IN. THE ADDRESS IS NOT READ BACK IN.
*
*
SC      EQU 10B         WCS SELECT CODE
WCSLD   NOP
        STF SC          INIT DIRECTION FF
        LDA =B-206      # OF MICROWORDS IN BUFFER
        STA CNT
*
* WRITE MICROWORDS OUT TO WCS
*
WRLP    DLD .OBF1,I     WRITE LOOP
        IOR WCSAD       "OR" IN WCS ADDRESS
        OTA SC          OUTPUT MICROWORDS
        OTB SC
        STC SC          WRITE PULSE
        ISZ .OBF1       POINT TO NEXT MICROWORD
        ISZ .OBF1
        LDA WCSAD
        ADA =B400       BUMP WCS ADDR BY 1
        STA WCSAD
        ISZ CNT
        JMP WRLP
*
* NOW READ THE MICROCODE BACK IN AND COMPARE
*
        CLA             1ST WCS ADDRESS = 0
        STA WCSAD
        LDA =B-206
        STA CNT
RDLP    STF SC          INIT DIRECTION FF
        LDA WCSAD       GET WCS ADDRESS
        OTA SC          OUTPUT ADDRESS TO WCS
        STF SC          REINIT FF
        LIA SC          INPUT MICROWORD
        LIB SC
        CPA .OBF2,I     DO COMPARES
        JMP BCOMP
```

```
        ISZ ERCNT       BUMP ERROR COUNT
BCOMP   ISZ .OBF2       POINT TO 2ND WORD
        CMA .OBF2,I
        JMP STWRD
        ISZ ERCNT       BUMP ERROR COUNT
STWRD   DST .IBF,I      STORE MICROWORDS
        ISZ .IBF        POINT TO NEXT POSITION
        ISZ .IBF
        ISZ .OBF2
        LDA WCSAD       GET WCS ADDRESS
        ADA =B400       BUMP IT BY 1
        STA WCSAD
        ISZ CNT
        JMP RDLP
*
*
CNT     OCT 0           LOOP COUNTER
WCSAD   OCT 0           WCS ADDRESS
ERCNT   OCT 0           ERROR COUNTER
.IBF    DEF IBUFF       INPUT BUFFER ADDRESS
.OBF1   DEF OBUFF       OUTPUT BUFFER ADDRESS
.OBF2   DEF OBUFF       ANOTHER ONE
* START OF "LOADS" MICROCODE
OBUFF   OCT 321,100130,301,170351,220,074457
        OCT 000,075717,017,101117,220,074457
        OCT 000,075717,017,101557,220,074457
        OCT 357,145617,017,101357,357,176557
        OCT 000,056461,177,126017,000,056517
        OCT 000,024461,177,126017,357,145657
        OCT 017,167157,220,044457,000,045117
        OCT 017,100557,220,044457,000,045117
        OCT 017,100517,321,102530,321,105370
        OCT 000,056461,177,126017,000,057357
        OCT 000,056461,177,124017,007,157357
        OCT 000,047157,000,047157,007,173657
        OCT 320,001171,000,057357,000,057357
        OCT 007,171617,320,000571,000,075736
        OCT 220,046457,301,141470,000,061417
        OCT 017,150157,004,161417,017,162544
        OCT 017,155057,301,104530,017,125217
        OCT 017,165057,017,127517,017,152544
        OCT 301,104530,017,126157,004,164557
        OCT 321,003571,000,024517,017,124157
        OCT 004,151517,017,154557,301,104530
        OCT 017,126544,017,126154,244,164542
        OCT 322,004271,325,044371,007,124517
        OCT 301,142530,321,101530,000,024517
        OCT 301,142530,321,101530,017,127014
        OCT 001,136517,017,142157,017,124255
        OCT 014,124504,017,140757,322,005131
        OCT 003,024517,017,142757,322,005331
        OCT 017,140157,003,024536,017,136776
        OCT 220,056457,301,141470,017,154517
        OCT 320,005631,346,001217,017,164757
```

```
        OCT 320,005771,346,001417,017,152557
        OCT 017,150157,003,061051,320,047171
        OCT 322,046371,010,043057,000,043057
        OCT 321,106670,017,124157,017,164517
        OCT 017,162557,015,037517,017,153457
        OCT 017,151417,347,120157,003,042757
        OCT 322,050131,037,124504,000,043057
        OCT 320,007031,017,162154,004,126557
        OCT 321,007431,340,000157,240,024517
        OCT 017,164151,244,124517,325,001031
        OCT 322,047671,017,136750,157,124504
        OCT 157,126544,000,061417,301,142530
        OCT 321,101570,017,164517,017,162557
        OCT 301,142530,321,101570
*   END OF "LOADS" MICROCODE
IBUFF   BSS 414B        INPUT BUFFER
        END WCSLD
```

Appendix I

This appendix contains listings for CDRVR and CALC. CDRVR is a special driver program, which was written to test the laser materials modeling program routine CALC on the AFIT 21MX computer. CDRVR provides all the inputs to CALC which would normally come from a potentiometer board on the AFWAL/MLPJ computer.

```
      6 AUG 82
      PROGRAM CDRVR
C
C  CDRVR IS A TEST DRIVER PROGRAM FOR THE SUBROUTINE CALC, A
C  ROUTINE WHICH CALCULATES THE REAL AND IMAGINARY PARTS OF
C  REFRACTIVE INDEX, A CHARACTERISTIC MEASURE OF LASER
C  MATERIALS. CDRVR IS USED TO DRIVE CALC ONLY FOR THE
C  PURPOSE OF MAKING TIMING MEASUREMENTS ON CALC. CALC IS ONE
C  ROUTINE OF SEVERAL USED IN A LASER MODELING PROGRAM
C  DEVELOPED BY AFWAL/MLPJ.
C
      COMMON IXO(150),IYO(150),B(30),G(30),IA(30),IQ(20)
      REAL BB(30),F,N,K
      INTEGER I,JJ
C
C  B(30) -- AN ARRAY CONTAINING PARAMETERS NORMALLY INPUT
C           FROM A 30-POT POTENTIOMETER BOARD
C  N -- THE REAL PART OF THE REFRACTIVE INDEX
C  K -- THE IMAGINARY PART OF THE REFRACTIVE INDEX
C  F -- RADIATION FREQUENCY
C  JJ -- THE NUMBER OF OSCILLATORS USED IN THE REFRACTIVE
C        INDEX CALCULATION
C
      DATA BB/1.0,800.0,1.0,1.0,800.0,1.0,1.0,800.0,1.0,
     * 1.0,800.0,1.0,1.0,800.0,1.0,1.0,800.0,1.0,1.0,800.0,
     * 1.0,1.0,800.0,1.0,0.0,400.0,100.0,2.0,0.5,0.0
      DATA JJ/8
C
C  COPY DATA FROM DUMMY BB ARRAY TO B ARRAY IN COMMON AREA
      DO 50 I=1,30
         B(I)=BB(I)
 50   CONTINUE
C
C  PERFORM CALCULATIONS FOR FREQUENCIES FROM 1000 TO 200 IN
C  STEPS OF 40 TO MAKE APPROXIMATELY 20 CALCULATIONS.
C
      DO 100 I=1000,200,-40
         F=1.0*I
         CALL CALC(B,JJ,F,N,K)
         WRITE(10,200)F,N,K
200      FORMAT(F20.9,2X,F20.9,2X,F20.9)
100   CONTINUE
      END
```

```
      SUBROUTINE CALC(B,JJ,F,C11,C12)
C
C  SUBROUTINE CALC CALCULATES THE REAL AND IMAGINARY PARTS OF
C  THE REFRACTIVE INDEX OF A LASER MATERIAL SIMULATED BY
C  PARAMETERS INPUT FROM A POTENTIOMETER BOARD. THE
C  CALCULATION IS PERFORMED BY EVALUATING EQUATIONS FOR
C  (N*N - K*K) AND (2*N*K) AND THEN SOLVING THESE TWO
C  EQUATIONS SIMULTANEOUSLY FOR N AND K (C11 AND C12)..
C
      COMMON IXO(150),IYO(150),B(30)
      REAL C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13
      INTEGER F2,J1,J2,J3
C
C  C1-C13 -- USED FOR INTERIM RESULTS IN EVALUATION OF THE
C            TWO LONG EQUATIONS
C  F2 -- FREQUENCY (F) SQUARED
C  J1-J3 -- INDICES OF ARRAY B USED TO PICK OUT THREE
C           DIFFERENT PARAMETERS -- DAMPING FACTOR, FREQUENCY
C           OF RESONANCE OF THE ITH OSCILLATOR, AND STRENGTH
C           OF RESONANCE
C
      F2=F*F
      C5=0.0
      C6=0.0
      DO 100 J=1,JJ
         J3=J*3
         J2=J1-1
         J1=J2-1
         C1=B(J1)*F
         C2=B(J2)*B(J2)
         C3=C2-F2
         C4=(B(J3)*C2)/(C3*C3+C1*C1)
         C5=C5+C3*C4
         C6=C6+C1*C4
 100  CONTINUE
C
      C7=B(27)*B(27)+F2
      C8=B(26)*B(26)
C  C9=N*N-K*K
      C9=B(28)+C6-B(29)*C8/C7
C  C10=2*N*K
      C10=C6+B(29)*B(27)*C8/(F*C7)
C  NOW SOLVE THESE 2 EQUATIONS FOR N AND K (C12 AND C13)
      C11=0.5*(-C9+SQRT(C9*C9+C10*C10))
      C12=SQRT(C9+C11)
      C13=SQRT(C11)
      RETURN
      END
      END$
```

Appendix J

This appendix contains listings for CALC, ACALC, and MCALC. The CALC in this listing is the same as in Appendix I except that the DO loop has been replaced by a call to ACALC. ACALC is an assembly language program which interfaces CALC to MCALC. MCALC is the microprogram which performs the function previously performed by the DO loop. CALC is again driven by CDRVR as shown in Appendix I.

```
      SUBROUTINE CALC(B,JJ,F,C11,C12)

C     SUBROUTINE CALC CALCULATES THE REAL AND IMAGINARY PARTS OF
C     THE REFRACTIVE INDEX OF A LASER MATERIAL SIMULATED BY
C     PARAMETERS INPUT FROM A POTENTIOMETER BOARD. THE
C     CALCULATION IS PERFORMED BY EVALUATING EQUATIONS FOR
C     (N*N - K*K) AND (2*N*K) AND THEN SOLVING THESE TWO
C     EQUATIONS SIMULTANEOUSLY FOR N AND K (C11 AND C12)..

          COMMON IXO(150),IYO(150),B(30)
          REAL C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,F2
          INTEGER J1,J2,J3
C
C     C1-C13 -- USED FOR INTERIM RESULTS IN EVALUATION OF THE
C               TWO LONG EQUATIONS
C     F2 -- FREQUENCY (F) SQUARED
C     J1-J3 -- INDICES OF ARRAY B USED TO PICK OUT THREE
C              DIFFERENT PARAMETERS -- DAMPING FACTOR, FREQUENCY
C              OF RESONANCE OF THE ITH OSCILLATOR, AND STRENGTH
C              OF RESONANCE
C
C     EVALUATE THE TWO EQUATIONS OVER JJ OSCILLATORS.
C     THIS IS DONE BY MICROPROGRAM MCALC WHICH IS INVOKED
C     BY THE ASSEMBLY LANGUAGE ROUTINE ACALC. RESULTS ARE
C     RETURNED IN C5 AND C6.
          F2=F*F
          C5=0.0
          C6=0.0
          CALL ACALC(JJ,F,F2,C5,C6)
          C7=B(27)*B(27)+F2
          C8=B(26)*B(26)
C     C9=N*N-K*K
          C9=B(28)+C6-B(29)*C8/C7
C     C10=2*N*K
          C10=C6+B(29)*B(27)*C8/(F*C7)
C     NOW SOLVE THESE 2 EQUATIONS FOR N AND K (C12 AND C13)
          C11=0.5*(-C9+SQRT(C9*C9+C10*C10))
          C12=SQRT(C9+C11)
          C13=SQRT(C11)
          RETURN
          END
          END$
```

```
ASMB,L
        NAM ACALC,7
        EXT .ENTR
        ENT ACALC
        COM IXO(150),IYO(150),B(60)
.JJ     BSS 1
..F     BSS 1
..F2    BSS 1
..C5    BSS 1
..C6    BSS 1
ACALC   NOP
        JSB .ENTR
        DEF .JJ
        DLD ..F         GET F AND F2 ADDRESSES
        DST .F          COPY INTO .F AND .F2
        DLD ..C5        GET C5 AND C6 ADDRESSES
        DST .C5         COPY INTO .C5 AND .C6
        LDX .B          PUT B ARRAY ADDRESS INTO X-REG
        LDY .JJ,I       PUT JJ INTO Y FOR LOOP COUNT IN MCALC
        LDA .TMP1       PUT TMP1 ADDRESS INTO A-REG
        LDA .TMP2       PUT TMP2 ADDRESS INTO B-REG
MCAL1   OCT 105620      INVOKE MCALC AT 1ST ENTRY POINT
.F      BSS 1           ADDRESS OF PARAMETER F
.F2     BSS 1           ADDRESS OF PARAMETER F2
FDIV    OCT 105060      INVOKE FLT PNT DIVIDE ROM ROUTINE
.C1C3   DEF C1C3        ARGUMENT FOR FDV
MCAL2   OCT 105621      INVOKE MCALC AT 2ND ENTRY POINT
.C5     BSS 1           OUTPUT PARAMETERS OF MCALC
.C6     BSS 1
        JMP ACALC,I     RETURN TO CALC
.B      DEF B           ADDRESS OF B ARRAY
.TMP1   DEF TMP1        ADDRESS OF TMP1
.TMP2   DEF TMP2        ADDRESS OF TMP2
TMP1    BSS 2           TMP1-TMP3 ARE WORKING
TMP2    BSS 2           LOCATIONS FOR MCALC
TMP3    BSS 2           TMP3 MUST FOLLOW TMP2
C1C3    BSS 2           HOLDS C1*C1+C3*C3
        END ACALC
```

```
MICMX,L,R                                   21MX
$CODE=%MCALC::20,REPLACE                    OBJECT TO DISK
        ORG  6000B
*************************************************************
*                                                           *
*                    MCALC  MICROPROGRAM                    *
*                                                           *
* THIS MICROPROGRAM IS A SUBSTITUTE FOR THE FOLLOWING       *
* LOOP IN THE FORTRAN SUBROUTINE CALC:                      *
*        DO 300 J=1,JJ                                      *
*           J3=J*3                                          *
*           J2=J1-1                                         *
*           J1=J2-1                                         *
*           C1=B(J1)*F                                      *
*           C2=B(J2)*B(J2)                                  *
*           C3=C2-F2                                        *
*           C4=(B(J3)*C2)/(C3*C3+C1*C1)                     *
*           C5=C5+C3*C4                                     *
*           C6=C6+C1*C4                                     *
*  300   CONTINUE                                           *
*                                                           *
* MCALC IS INVOKED BY FIRST CALLING AN ASSEMBLY             *
* LANGUAGE ROUTINE CALLED ACALC FROM CALC AT THE            *
* POINT WHERE THE ABOVE LOOP RESIDED. ACALC THEN            *
* INVOKES THE MICROPROGRAM WITH THE FOLLOWING               *
* INSTRUCTIONS:                                             *
*                                                           *
*       LDX .B       PUT B ARRAY ADDR INTO X-REG            *
*       LDY .JJ,I    PUT JJ INTO Y FOR LOOP COUNTER         *
*       LDA .TMP1    PUT TMP1,TMP2,TMP3                     *
*       LDA .TMP2    TMP1,TMP2,TMP3 ARE DEFINED AS          *
*                    "BSS 2". TMP3 MUST IMMEDIATELY         *
*                    FOLLOW TMP2 TO PASS ITS ADDRESS.       *
* MCAL1 OCT 105620   INVOKE MCALC AT 1ST ENTRY POINT        *
* .F    BSS 1        ADDR OF F                              *
* .F2   BSS 2        ADDR OF F2   (F2=F*F)                  *
* FDIV  OCT 105060   INVOKE FLT PNT DIVIDE ROM ROUTINE      *
* .C1C3 DEF C1C3     ARGUMENT FOR FLT PNT DIVIDE            *
* MCAL2 OCT 105621   INVOKE MCALC AT 2ND ENTRY POINT        *
* .C5   BSS 1        OUTPUT PARAMETERS OF MCALC             *
* .C6   BSS 1                                               *
*                                                           *
* MCALC IS INVOKED TWICE AT TWO DIFFERENT ENTRY             *
* POINTS. THE REASON FOR THIS IS THAT A FLOATING            *
* POINT DIVIDE MUST BE PERFORMED IN THE MIDDLE OF           *
* MCALC, BUT THE DIVIDE ROUTINE WILL NOT FIT IN WCS,        *
* SO THE ROM ROUTINE IS USED. THIS REQUIRES A RETURN        *
* TO ACALC TO INVOKE THE ROM ROUTINE. MCALC IS THEN         *
* REENTERED TO COMPLETE ITS OPERATION.                      *
*************************************************************

MPYX      EQU              %0246   ROM FLT PNT MPYX ROUTINE
FLD       EQU              %7031   ROM FLT PNT LOAD ROUTINE
PACK      EQU              %7052   ROM FLT PNT PACK ROUTINE
```

```
START1   JMP                    MCALC1  GO TO 1ST ENTRY POINT
START2   JMP                    MCALC2  GO TO 2ND ENTRY POINT

*************************************************************
* THE FOLLOWING RETURN TABLE IS USED TO JUMP BACK TO        *
* THE MAIN MICROPROGRAM FROM SUBROUTINES FMPY,FADD OR       *
* FSUB. NORMAL RETURNS CANNOT BE MADE BECAUSE THE           *
* RETURN ADDRESS IS LOST WHEN FMPY,FADD OR FSUB CALL        *
* OTHER ROUTINES. THIS JUMP TABLE IS USED AS FOLLOWS:       *
* RTNTABLE IS LOCATED AT 6002 AND CONTAINS A JUMP TO        *
* THE 1ST LOCATION FOLLOWING THE 1ST CALL TO FMPY.          *
* BEFORE FMPY IS CALLED, THE IR-REG IS LOADED WITH          *
* VALUE 2 IN BITS 0-3. THE RETURN FROM FMPY IS VIA A        *
* "JMP J74" USING BITS 4-7 OF THE IR, SO THE RETURN         *
* INDEX FOR A FMPY AND A SUBSEQUENT FADD OR FSUB CAN        *
* BE LOADED INTO THE IR AT THE SAME TIME.                   *
*************************************************************
RTNTABLE JMP                    RTNPNT1   TABLE OF JUMPS TO
         JMP                    RTNPNT2   RETURN POINTS FROM
         JMP                    RTNPNT3   SUBROUTINE CALLS.
         JMP                    RTNPNT4   BEFORE JUMPING TO A
         JMP                    RTNPNT5   SUBROUTINE THE LOWER 4
         JMP                    RTNPNT6   BITS OF THE RTNTABLE
         JMP                    RTNPNT7   JMP ENTRY ARE LOADED
         JMP                    RTNPNT8   INTO THE IR. THE
         JMP                    RTNPNT9   SUBROUTINE DOES A "JMP
         JMP                    RTNPNT10  J30 RTNTABLE" TO
         JMP                    RTNPNT11  RETURN TO A CALLER.
                                          THE "J30" REPLACES THE
                                          LOWER 4 BITS OF THE
                                          JMP ADDR WITH THE 4 IR
                                          BITS
DEBUG    JMP                    DEBUG     DUMMY ENTRY FOR DEBUG
*************************************************************
* SET UP CALLING PARAMETERS.                                *
* SCRATCH REGISTERS:    JJ    --> Y-REG                     *
*                       .B    --> X-REG                     *
*                       .TMP1 --> S4                        *
*                       .TMP2 --> S8                        *
*                       .TMP3 --> S12                       *
*************************************************************
MCALC1            PASS S   P      SAVE P IN S
                  PASS S4  A      USE S4 AS POINTER TO TMP1
                  PASS S8  B      USE S8 AS POINTER TO TMP2
                  INC  S12 S8     USE S12 AS POINTER TO TMP3
                  INC  S12 S12    NOTE TMP3 IS AT TMP2+2
*************************************************************
* CALCULATE C1. C1=GAMMA1(J)*F                              *
* NOTE THAT GAMMA1(J), NU(J), AND RHO(J) ARE ELEMENTS       *
* OF THE B ARRAY, AND ARE ARRANGED IN THE ARRAY IN          *
* THAT ORDER. I.E., B(1)=GAMMA1(1), B(2)=NU(1), B(3)=       *
* RHO(1), B(4)=GAMMA1(2), B(5)=NU(2), B(6)=RHO(2) ...       *
* B(22)=GAMMA1(8), B(23)=NU(8), AND B(24)=RHO(8).           *
```

```
*********************************************************
LOOP     READ       INC  M   X      READ GAMMA1 ELEMENT FROM B
                    INC  X   X      POINT TO 2ND GAMMA1 WORD
                    PASS A   TAB    PUT 1ST GAMMA1 WORD INTO A
         READ       INC  M   X      READ 2ND GAMMA1 WORD
                    INC  X   X      POINT TO NU ELEMENT OF B
                    PASS B   TAB    PUT 2ND GAMMA1 WORD INTO B
         READ       INC  PNM P      READ F ADDR. POINT TO .F2
                    PASS S3  TAB    F ADDR INTO S3 FOR MPY
         IMM        CMLO S1  %374   LO 4 MAP TO "JMP RTNPNT1"
                    PASS IR  S1
         JMP                FMPY   GAMMA1(J)*F=C1 RETURN IN A
RTNPNT1       MPCK  INC  M   S4     POINT M AT TMP1
         WRTE       PASS TAB A      WRITE 1ST C1 WORD TO TMP1
                    INC  S3  S4     POINT S3 TO 2ND TMP1 WORD
              MPCK  INC  M   S3     NOW, SO DOES M
         WRTE       PASS TAB B      WRITE 2ND C1 WORD TO TMP2
*********************************************************
* CALCULATE C2.  C2=NU(J)*NU(J)                         *
*********************************************************
                    PASS S3  X      NU(J) ADDR INTO S3
         READ       INC  M   X      READ NU ELEMENT FROM B
                    INC  X   X      POINT TO 2ND WORD OF NU
                    PASS A   TAB    PUT 1ST WORD OF NU INTO A
         READ       INC  M   X      READ 2ND WORD OF NU
                    INC  X   X      POINT TO RHO ELEMENT OF B
                    PASS B   TAB    PUT 2ND WORD OF NU INTO B
         IMM        CMLO S1  %253   LO 4 MAP TO "JMP RTNPNT2"
                    PASS IR  S1     HI 4 MAP TO "JMP RTNPNT3"
         JMP                FMPY   NU(J)*NU(J) RETURNS IN AB
RTNPNT2       MPCK  INC  M   S8     POINT M AT TMP2
         WRTE       PASS TAB A      WRITE 1ST C2 WORD TO TMP2
                    INC  S3  S8     POINT S3 TO 2ND TMP2 WORD
              MPCK  INC  M   S3     AND M ALSO
         WRTE       PASS TAB B      WRITE 2ND C2 WORD
*********************************************************
* CALCULATE C3.  C3=C2-F2                               *
*********************************************************
         READ       INC  PNM P      READ .F2. POINT TO FDV INS
                    PASS S3  TAB    PUT F2 ADDR (.F2) INTO S3
         JMP                FADDSUB C2-F2 RETURNS IN A/B
RTNPNT3       MPCK  INC  M   S12    POINT M AT TMP3
         WRTE       PASS TAB A      WRITE 1ST C3 WORD TO TMP3
                    INC  S3  S12    S3 POINTS TO TMP3+1
              MPCK  INC  M   S3     AND SO DOES M
         WRTE       PASS TAB B      WRITE 2ND C3 WORD TO TMP3
*********************************************************
* CALCULATE C4.  C4=(RHO(J)*C2)/(C3*C3+C1*C1)           *
*********************************************************
         READ       INC  M   S8     READ 1ST WORD OF C2 (TMP2)
                    INC  S3  S8     S3 POINTS TO 2ND WORD
                    PASS A   TAB    1ST WORD OF C2 INTO A-REG
         READ       INC  M   S3     READ 2ND WORD
                    PASS S3  X      PASS S3 AT RHO ELEMENT OF
```

```
                        INC   X    X       INC X BY 2 TO POINT AT
                        INC   X    X       NEXT GAMMA
                        PASS  B    TAB     2ND WORD OF C2 INTO B-REG
              IMM       CMLO  S1   %371    LO 4 MAP TO "JMP RTNPNT4"
                        PASS  IR   S1
              JMP                  FMPY    RHO(J)*C2 RETURNS IN A/B
RTNPNT4       MPCK      INC   M    S8      POINT M AT TMP2
              WRTE      PASS  TAB  A       WRITE 1ST WORD RHO(J)*C2
                        INC   S3   S8      POINT S3 TO 2ND WORD TMP2
              MPCK      INC   M    S3      AND M ALSO
              WRTE      PASS  TAB  B       WRITE 2ND WORD RHO(J)*C2
              READ      INC   M    S12     READ 1ST WORD OF C3
                        INC   S3   S12     POINT S3 AT 2ND WORD
                        PASS  A    TAB     PUT 1ST WORD INTO A-REG
              READ      INC   M    S3      READ IN 2ND WORD OF C3
                        PASS  S3   S12     POINT S3 BACK AT 1ST WORD
                        PASS  B    TAB     PUT 2ND WORD INTO B-REG
              IMM       CMLO  S1   %370    LO 4 MAP TO "JMP RTNPNT5"
                        PASS  IR   S1
              JMP                  FMPY    C3*C3 RETURNS IN A/B
RTNPNT5                 INC   S3   P       POINT S3 AT .C1C3 ADDRESS
              READ      INC   M    S3      READ C1C3 ADDRESS
                        PASS  S3   TAB     AND PUT INTO S3
              MPCK      INC   M    S3      POINT M AT C1C3
              WRTE      PASS  TAB  A       WRITE 1ST WORD OF C3*C3
                        INC   S3   S3      POINT S3 AT 2ND WORD C1C3
              MPCK      INC   M    S3      AND M ALSO
              WRTE      PASS  TAB  B       WRITE 2ND WORD OF C3*C3
              READ      INC   M    S4      READ IN 1ST WORD OF C1
                        INC   S3   S4      POINT S3 AT 2ND WORD
                        PASS  A    TAB     PUT 1ST WORD INTO A-REG
              READ      INC   M    S3      READ IN 2ND WORD OF C1
                        PASS  S3   S4      POINT S3 BACK AT 1ST WORD
                        PASS  B    TAB     PUT 2ND WORD INTO B-REG
              IMM       CMLO  S1   %147    LO 4 MAP TO "JMP RTNPNT6"
                        PASS  IR   S1      HI 4 MAP TO "JMP RTNPNT7"
              JMP                  FMPY    C1*C1 RETURNS IN A/B
RTNPNT6                 INC   S3   P       POINT S3 AT .C1C3 ADDRESS
              READ      INC   M    S3      READ C1C3 ADDRESS
              STFL      PASS  S3   TAB     INTO S3. STFL FOR NEXT ADD
              JMP                  FADDSUB C1*C1+C3*C3 RETURNS IN AB
RTNPNT7                 INC   S3   P       POINT S3 AT .C1C3 ADDRESS
              READ      INC   M    S3      READ C1C3 ADDRESS
                        PASS  S3   TAB     AND PUT INTO S3
              MPCK      INC   M    S3      AND INTO M
              WRTE      PASS  TAB  A       WRITE 1ST WORD C1*C1+C3*C3
                        INC   S3   S3      POINT S3 AT WORD 2 OF C1C3
              MPCK      INC   M    S3      AND M ALSO
              WRTE      PASS  TAB  B       WRITE 2ND WORD C1*C1+C3*C3
              READ      INC   M    S8      READ 1ST WORD RHO(J)*C2
                        INC   S3   S8      POINT AT 2ND WORD
                        PASS  A    TAB     PUT 1ST WORD INTO A-REG
              READ      INC   M    S3      READ 2ND WORD
              RTND      PASS  B    TAB     PUT 2ND WORD INTO B
```

```
*********************************************************
* AT THIS POINT EVERYTHING IS SET UP FOR THE DIVIDE     *
* OF RHO(J)*C2 BY C1*C1+C3*C3. RETURN TO THE ASSEMBLY*
* LANGUAGE ROUTINE TO INVOKE THE FLOATING POINT         *
* DIVIDE ROUTINE. RETURN TO MICROCODE AT MCALC2 WITH *
* THE RESULT IN A/B REGS AND OTHER REGS INTACT.         *
*********************************************************
MCALC2          MPCK INC   M    S8      POINT M AT 1ST WORD TMP2
          WRTE       PASS TAB A       1ST WORD OF C4 INTO TMP2
                     INC  S3   S8      POINT S3 AT 2ND TMP2 WORD
                MPCK INC  M    S3      AND M ALSO
          WRTE       PASS TAB B       2ND WORD OF C4 INTO TMP2
*********************************************************
* CALCULATE C5.  C5=C5+C3*C4                            *
*********************************************************
                     PASS S3   S12     ADDRESS OF C3 INTO S3
          IMM        CMLO S1   %105    LO 4 MAP TO "JMP RTNPNT8"
                     PASS IR   S1      HI 4 MAP TO "JMP RTNPNT9"
          JMP                  FMPY    C3*C4 RETURNS IN A/B
RTNPNT8         MPCK INC  M    P       READ C5 ADDRESS
                STFL PASS S3   TAB     INTO S3. STFL FOR NEXT ADD
           JMP                 FADDSUB C5+C3*C4 RETURNS IN A/B
RTNPNT9    READ       INC  PNM P       GET C5 ADDR. POINT C6 ADDR
                      PASS S3   TAB     AND PUT INTO S3
                MPCK INC  M    S3      C5 ADDRESS INTO M
          WRTE       PASS TAB A       1ST WORD OF C5 STORED
                     INC  S3   S3      POINT TO 2ND WORD
                MPCK INC  M    S3      AND M ALSO
          WRTE       PASS TAB B       2ND WORD OF C5 STORED
*********************************************************
* CALCULATE C6.  C6=C6+C1*C4                            *
*********************************************************
          READ       INC  M    S4      READ IN 1ST WORD OF C1
                     INC  S3   S4      POINT S3 AT 2ND WORD OF C1
                     PASS A    TAB     1ST WORD OF C1 INTO A-REG
          READ       INC  M    S3      READ IN 2ND WORD OF C1
                     PASS S3   S8      POINT S3 AT C4
                     PASS B    TAB     2ND WORD OF C1 INTO B-REG
          IMM        CMLO S1   %043    LO 4 MAP TO "JMP RTNPNT10"
                     PASS IR   S1      HI 4 MAP TO "JMP RTNPNT11"
          JMP                  FMPY    C1*C4 RETURNS IN A/B
RTNPNT10  READ       INC  M    P       READ C6 ADDRESS
                STFL PASS S3   TAB     INTO S3. STFL FOR NEXT ADD
          JMP                  FADDSUB C6+C1*C4 RETURNS IN A/B
RTNPNT11  READ MPCK  INC  PNM P       GET C6 ADR. POINT TO NEXT
                     PASS S3   TAB     PUT C6 ADDRESS INTO S3
                     INC  M    S3      AND INTO M
          WRTE       PASS TAB A       1ST WORD OF C6 STORED
                     INC  S3   S3      POINT TO 2ND WORD
                MPCK INC  M    S3      M ALSO
          WRTE       PASS TAB B       2ND WORD OF C6 STORED
                     DEC  Y    Y       DECREMENT LOOP COUNT
          JMP  CNDX  TBZ       RTNMAC  IF DONE, RETURN
                     PASS P    S       POINT P AT .F & DO AGAIN
```

```
        JMP                     LOOP
RTNMAC      RTN                         RETURN TO ACALC
*************************************************************
* FLOATING POINT MULTIPLY ROUTINE. THIS ROUTINE IS          *
* TAKEN FROM APPENDIX E OF THE HP MICROPROGRAMMING          *
* 21MX COMPUTERS OPERATING AND REFERENCE MANUAL. THIS       *
* ROUTINE ALSO RESIDES IN CONTROL STORE ROM, BUT IT         *
* IS NECESSARY TO REPRODUCE IT IN WCS TO AVOID THE          *
* PROBLEM OF LEVELED SUBROUTINE CALLS IN THE M-SERIES       *
* COMPUTER. FMPY HAS BEEN MODIFIED SLIGHTLY TO HANDLE       *
* THE ARGUMENT ADDRESS IN REGISTER S3 RATHER THAN P.        *
* THE RETURN TO THE CALLING ROUTINE HAS BEEN MODIFIED       *
* TO A "JMP J30 RTNTABLE" AS DISCUSSED AT "RTNTABLE".       *
* ALSO, INDIRECT ADDRESSING IS NOT SUPPORTED.               *
*************************************************************

FMPY    READ        INC   M   S3      READ 1ST PARAMETER WORD
        JSB                   FLD     STORE ARGS IN SCRATCH REGS
                    INC   S9  S9
                    PASS  L   S5      FORM EXP1+EXP2+1
                    ADD   S9  S9      AND SAVE IN S9
        R1          PASS  A   S10     FORM (WORD1 LOBITS)/2 IN A
                    PASS  S2  S7      PASS WORD2 HIBITS INTO S2
        JSB                   MPYX    JMP TO MPY SUB & RTN WITH
                    PASS  S5  B       HIBITS IN B & SAVE IN S5
                    PASS  S2  S11     PASS WORD1 HIBITS INTO S2
                    PASS  S11 A       LOBITS INTO A. SAVE INTO S
            R1      PASS  A   S6      FORM (WORD2 LOBITS)/2 IN A
        JSB                   MPYX    JMP TO MPY SUB & RTN WITH
                    PASS  L   A       LOBITS IN A & PASS INTO L
                    ADD   A   S11     ADD BOTH LOBITS. CHK FOR C
        JMP   CNDX  COUT  RJS *+2     (ELSE TRUNCATE DIGITS)
                    INC   B   B       IF COUT, BUMP HIBITS
                    PASS  L   B       ADD HIBITS & SAVE IN S11
                    ADD   S11 S5
                    PASS  A   S7      PASS WORD2 HIBITS INTO A
        JSB                   MPYX    JMP TO MPY SUB & RTN WITH
            R1      PASS  A   A       LOBITS IN A. SAVE LOBITS/2
            COV     PASS  L   A       ADD LOBITS/2 TO HIBITS SUM
        ENV L1      ADD   A   S11     SHIFT L1 TO REORIENT
        JMP   CNDX  AL15  RJS *+3     CHECK FOR CARRY INTO OR
        JMP   CNDX  OVFL      *+4     BORROW FROM HIBITS &
                    DEC   B   B       ADJUST ACCORDINGLY
        JSB                   PACK
        JMP   J30             RTNTABLE RTN TO MAIN MICROPROGRAM
                    INC   B   B       CAN'T OVERFLOW FROM HIBITS
        JSB                   PACK
        JMP   J30             RTNTABLE RTN TO MAIN MICROPROGRAM
*************************************************************
* FLOATING POINT ADD SUBTRACT ROUTINE. THIS ROUTINE         *
* IS TAKEN FROM APPENDIX E OF THE HP MICROPROGRAMMING       *
* 21MX COMPUTERS OPERATING AND REFERENCE MANUAL. IT         *
* ALSO RESIDES IN CONTROL STORE ROM BUT IS DUPLICATED       *
* IN WCS TO AVOID THE PROBLEM OF LEVELED SUBROUTINE         *
```

```
* CALLS IN THE M-SERIES. FADD HAS BEEN MODIFIED TO    *
* ALLOW A PARAMETER ADDRESS IN SCRATCH REGISTER S3     *
* RATHER THAN THE P REGISTER. INDIRECT ADDRESSING IS   *
* NOT USED, SO THE CALL TO "INDIRECT" IS OMITTED.      *
* ALSO, SCRATCH REGISTER S2 IS USED IN PLACE OF S8 TO  *
* FREE S8 FOR USE IN THE MAIN PROGRAM. THE RETURN TO   *
* THE CALLING ROUTINE HAS ALSO BEEN MODIFIED TO A      *
* "JMP J74 RTNTABLE" AS DISCUSSED AT "RTNTABLE".       *
*******************************************************

FADD     READ       INC  M    S3      READ 1ST PARAMETER WORD
         JSB                  FLD     UNPACK WORDS INTO SCR REGS
                    PASS B    S7      CHECK FOR WORD2=0
         JMP  CNDX  TBZ  RJS  *+2     IF NOT, CONTINUE
         IMM        LOW  S5   %200    IF SO, MAKE EXP MOST NEG
                    PASS      S11     CHECK FOR WORD1=0
         JMP  CNDX  TBZ  RJS  *+2     IF NOT, CONTINUE
         IMM        LOW  S9   %200    IF SO, MAKE EXP MOST NEG
         JMP  CNDX  FLAG      DIFR    IF DOING ADD,SKIP AHEAD
                    CMPS B    B       FORM 2-COMP OF HIBITS IN B
                    CMPS S6   S6      FORM 2-COMP OF LOBITS
                    INC  S6   S6      OF WORD2
         JMP  CNDX  COUT RJS  DIFR    IF COUT OCCURS
                    INC  B    B       BUMP HIBITS
         JMP  CNDX  AL15 RJS  DIFR    CHECK SIGN   IF POS,JUMP
              L1    PASS      B       IF NEG,CHECK FOR MOST
         JMP  CNDX  TBZ  RJS  DIFR    NEG # (100...)
              R1    PASS B    B       IF SO, SHIFT BACK (010...)
                    INC  S5   S5      & BUMP EXP
DIFR                PASS A    S6
                    PAS  L    S5      FIND DIFF IN EXPS
              CLFL  SUB  S2   S9      & STORE IN S2, FLAG=0
         JMP  CNDX  TBZ       ADD2    IF DIFF=0, JMP TO ADD STEP
         JMP  CNDX  AL15      RVRS    IF NEG, WORD2>WORD1
                    CMPS S2   S2      FORM -DIFF
                    INC  S2   S2      & STORE -DIFF IN S2
         JMP                  SWAMPCHK
RVRS                PASS L    B       HOLD B IN L
                    PASS B    S11     WORD1<WORD2, FILL IN B,A
                    PASS A    S10     WITH S11,S10
                    PASL S11          ALSO FILL S11,S10,S9
                    PASS S10  S6      WITH B,S6,S5
                    PASS S9   S5
SWAMPCHK IMM        LOW  L    %350    FORM -30B8 IN L
                    SUB       S2      IF -DIFF>-31,RTN WITH
*                                     LARGER #
         JMP  CNDX  AL15      OUT     JMP TO RESTORE A,B
SHIFT    ARS  R1    PASS B    B       NOW START SHIFT LOOP
                    INC  S2   S2      INC COUNTER
         JMP  CNDX  TBZ  RJS  SHIFT   LOOP UNTIL DONE
ADD2          COV   PASS L    S10     PASS LOBITS INTO L
                    ADD  A    A       ADD & CHECK FOR COUT
         JMP  CNDX  COUT RJS  *+3     IF NOT, JUMP
         IMM        HIGH L    %0      CLR L(15) FOR OVFL
```

```
            ENV         INC   B     B       IF SO,INC HIBITS & ENABLE
*                                           OVERFLOW
                  CLFL  PASS  L     S11     FLAG=0
            ENV         ADD   B     B       ADD HIBITS & ENABLE OVFL
            JMP   CNDX  OVFL  RJS   PKSUB   IF NO OVFL, RETURN
            JMP   CNDX  AL15        *+2     OVFL IMPLIES SIGN CHANGE
                  STFL                      SO FLAG=U IF AL15=0
            LWF   R1    PASS  B     B       DO FULL WORD SHIFT
            LWF   R1    PASS  A     A       USING FLAG REG TO INJECT
*                                           SIGN
                        INC   S9    S9      BUMP EXP
PKSUB       JSB                     PACK    REPACK A,B REGS
            JMP   J74               RTNTABLE RTN TO MAIN MICROPROGRAM
OUT                     PASS  B     S11     PASS MUCH LARGER WORD INTO
*                                           B,A
                        PASS  A     S10
            JSB                     PACK
            JMP   J74               RTNTABLE RTN TO MAIN MICROPROGRAM
            END
```

Vita

Captain Gary A. Schoon was born on June 10, 1950 in Washburn, Illinois. He graduated from Metamora Township High School, Metamora, Illinois in 1968. After attending the University of Illinois for one year, he entered the United States Air Force in 1969. He served as an electronics technician at various assignments within the United States and overseas. In 1974 he returned to the University of Illinois under the Airman's Education and Commissioning Program and received a Bachelor of Science degree in Computer Engineering. After receiving his commission at the Air Force Officer Training School in 1977, he was assigned to the North American Aerospace Defense Command in Colorado Springs, Colorado, where he served as a systems analyst. He entered the Air Force Institute of Technology in 1981.

Permanent address: 4530 Debonair Circle
Colorado Springs, Colorado 80917

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/82D-31	2. GOVT ACCESSION NO. A124853	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) APPLICATIONS DIRECTED MICROPROGRAMMING ON A MINICOMPUTER SYSTEM		5. TYPE OF REPORT & PERIOD COVERED MS THESIS 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) GARY A. SCHOON, Capt, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, OH 45433		12. REPORT DATE December 1982 13. NUMBER OF PAGES 166
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES
Approved for public release; IAW AFR 190-17

Approved for public release: IAW AFR 190-17.
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB OH 45433

JAN 1983

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Microprogramming
Computer Architecture
Tuning
HP 21MX Computer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
The use of microprogramming to improve the performance of application programs was investigated. The application programs used in the study were from various research laboratories at Wright-Patterson Air Force Base, Ohio. The user-microprogrammable Hewlett-Packard (HP) 21MX minicomputer was used for the investigation.

Two application programs were chosen as candidates for microprogramming, a wind tunnel stress analysis program and a laser materials modeling program. The programs were analyzed to determine where microprogramming should be applied

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

using an activity profile generator program. The microcode for the programs was implemented, and the speed improvement measurements of the resultant programs were made.

The study further looked at the feasibility of automating the microprogramming tuning process on the HP 21MX computer. Approaches to automatically selecting program segments for microprogramming and automatically synthesizing the microcode were discussed.

END

FILMED

3-83

DTIC